Getting Started with MTEX for EBSD analysis

Written by J. Hiscocks for version 5.5.2 of MTEX and MATLAB 2020b
Based heavily on the documentation published online by Ralf Hielscher
Manual Revision 6
For easier navigation of this document, open the Navigation pane. This document was mostly written with magnesium (and HCP) examples, so if you're working with another material keep in mind that commands and Miller indexes may not be interchangeable between HCP, Cubic, and other structures.
I've also created an app that lets you do many common MTEX functions graphically and generate the corresponding command script for later use! It doesn't have the functionality of all the commands in this document, but it can be helpful if you're looking for a quick plot. Download it via my ResearchGate page.

Abstract:
MTEX is a free add-on to the base installation of MATLAB (no packages required) and can be used to analyse EBSD and XRD (pole figure input) data. In comparison to the Bruker and Channel 5/ HKL software packages it has the advantage of allowing you to create any sort of map or analysis supported by your input data, no matter how complex. You can also automate tasks with scripts which can be handy when you want to repeat analysis or make the same figure for multiple scans or datasets. With MTEX, batch analysis is straightforward. Another advantage of using MTEX is that the supporting MATLAB commands let you customize the plot font, colours, and resolution. However, MTEX is command line operated, and thus has a much steeper learning curve to get started.
While there are various tutorials available online, the purpose of this document is to help a first-time user do some basic analysis tasks and understand how the MTEX commands are structured to help you construct your own. This document is not intended to be a comprehensive overview of all functions and is intended mainly to be a 'getting started' guide to analysing and creating figures from EBSD data.

Additional help:
1. For the official webpage start at http://mtex-toolbox.github.io/documentation.html
2. For help try looking in https://github.com/mtex-toolbox/mtex/discussions
3. or posting a question there AFTER YOU HAVE CHECKED THIS DOCUMENT AND THE WEB.
4. For scripts, try https://gist.github.com/ search for #mtexScript

Table of Contents

Quick start:

I get it, no one wants to read the whole 97 pages- you want to get your analysis done and move on. The following sections are ESSENTIAL to understanding what you're doing and creating valid results.

- Working through is key to making sure your data is oriented correctly when it's loaded into MTEX. Getting this part wrong can completely invalidate your results.
- Understanding the section is critical to understanding which commands to use for which types of actions. If you want to do more than copy and paste commands, you must read and understand this section.

Once you cover these two sections you can probably skip to whatever input you're most interested in.

Getting more familiar with MTEX

The best way to really start learning this software is to work through the example code line by line in MATLAB, monitoring the variables by looking at their size in the workspace, and looking at what's stored in the variable (double click on the variable name). You can start even before you have your own data since MTEX has you can use when getting started.

Installing MTEX and initial setup

Download MTEX and extract the folder to somewhere convenient. Open MATLAB AS AN ADMINISTRATOR (right click on the MATLAB icon to get this option). In MATLAB, navigate to the directory containing MTEX. You should see an entry named `startup_mtex.m` in the current folder window. Type `startup_mtex` in the MATLAB command window. If you get an error, wait a minute and retry. Pre-existing MTEX installations will be removed. Click on 'install MTEX for future sessions', and you'll be able to skip this step next time. The MTEX header should display in the command line area and tell you the version number you installed. If you don't see this header when starting MATLAB, MTEX is not installed or active.

Optionally, some MTEX settings can be configured to customize it for your own data. Each time you install MTEX or change the version you'll need to repeat this customization. To do this, open MATLAB AS AN ADMINISTRATOR and edit the settings by typing `open mtex_settings.m`

Changes I generally make to MTEX defaults (note the line numbers may change slightly):

```
%change the default plotting of the coordinate axes to match my data's
spatial orientation (about line 30)
setMTEXpref('xAxisDirection','west');
%I usually don't want the micron bar on EBSD maps (line 45)
setMTEXpref('showMicronBar','off')
%I plot PF on multiple planes, so the default annotation is not
helpful and I disable it,  by uncommenting line 57 to prevent
annotations
%I prefer less padding on multiple PF (lines 61 and 62)
setMTEXpref('outerPlotSpacing',0);
setMTEXpref('innerPlotSpacing',0);
```

I strongly recommend entering your x-y axis preferences for your system once you know them. Otherwise you need to remember to set them every session. I discuss how to figure out what to use in . Now save the .m file, exit and restart MATLAB.

Input and Output

Your data was probably acquired on an instrument that's separate from the computer you'll use to do the analysis. Even if they're on the same machine, your data needs to be moved between the instrument (acquisition system) software and MTEX (running on MATLAB). Since these two softwares won't directly communicate, you need to make sure that MTEX is informed of all the essential information to orient your data correctly. This is covered in .

Exported data format and some advice

Transferring data from your equipment into MTEX is often done with it as a .ctf (channel text file). Most data acquisition systems will export data in this format but be aware this format only saves one line of text for each measured point. Be sure to save your acquired data in your acquisition software's native data format as well, which will include images, compositional data etc.

Note that the .ctf file can be opened as a text file (using notepad in windows) but this is usually slow since the files are quite large. An example of this is shown as Figure 1. The header of your text file is worth reading, it may have valuable information on how your system saves data, or about the step size used for the scan. The text files can also be edited in Excel or MATLAB and converted back into a text file, making .ctf a good data format for being able to see directly how your sample data is handled and edit it as required.

If your system can't save into a format MTEX knows, you can manually format your data as a text file, with data in columns, like that shown.

```
Channel Text File
Prj unnamed
Author    [Unknown]
JobMode Grid
XCells    1506
YCells    1301
XStep     9.998745245E-1
YStep     9.973563196E-1
AcqE1     0
AcqE2     0
AcqE3     0
Euler angles refer to Sample Coordinate system (CS0)!       Mag       9.3E1      Coverage 100       Device    0          KV
          2E1        TiltAngle   70         TiltAxis    0
Phases    1
3.209;3.209;5.21     9E1;9E1;1.2E2        Magnesium          9         194
Phase     X          Y          Bands      Error      Euler1     Euler2     Euler3     MAD        BC         BS
0         0.0000     0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -0.9999    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -1.9997    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -2.9996    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -3.9995    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -4.9994    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -5.9992    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -6.9991    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -7.9990    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -8.9989    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
0         -9.9987    0.0000     0          3          0.0000     0.0000     0.0000     0.0000     0          255
```

Figure 1: An example of a .ctf file, opened as a text file

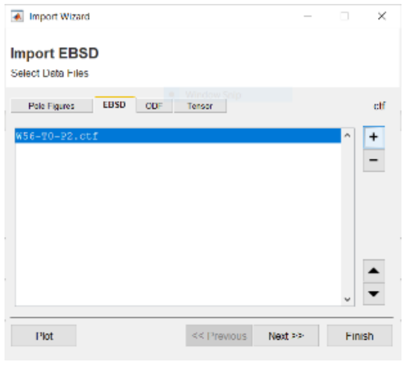
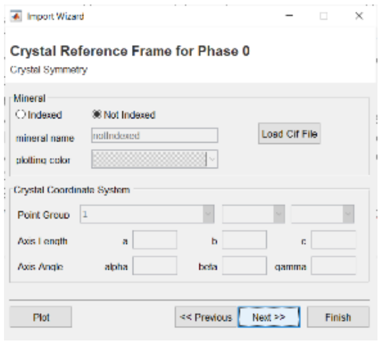
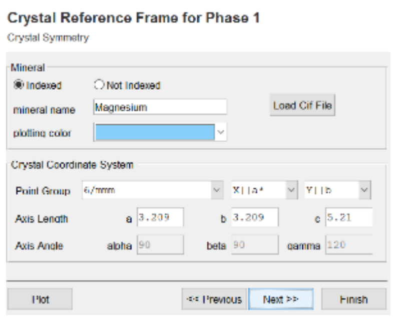
Taking a look at Figure 1, you can see that the Bruker acquisition software inserted padding (zero solutions). This software adds these zero solutions as a border around any non-full screen EBSD map which can inflate file sizes significantly and bog down your computer. This padding of zero solutions can also interfere with analysis (such as indexing percentages), so after importing it you can remove the padding prior to analysis with the command `ebsd=ebsdPADDED('indexed')` This command tells the software to take the indexed points of `ebsdPADDED` and copy them to a new variable named `ebsd`. Just like MATLAB, the right side of the command is the input, and the left side of the command is the output.

If for some reason you didn't want to remove the zero solutions this way, you could crop the map as described in section .

To import data

1. Click on "import EBSD data" link that shows up in the MTEX header when you start MATLAB, or use the command `import_wizard('EBSD');`
2. In the dialog box that appears, click on + at the right of the dialog box and select your .ctf file (see Figure 2). Click next.
3. Based on the input file MTEX will now show options for each phase in your data (starting with not indexed). In Figure 3, information for the "Not indexed' data is shown. Click "next"
4. Information about Magnesium is now shown, see Figure 4. Click "next".

5. Figure 5 shows the dialog for verifying the spatial orientation of the map is correct (see). Use the plot button in the bottom left of the dialog to check your import settings and change coordinate setup as necessary until correct (See Figure 6).
6. Verify the Euler angles (see) by clicking on the plot and reading the data cursor.
7. Select 'import to a workspace variable' (see Figure 7), and leave the default name of 'ebsd'. Click finish.

 Import Wizard **Import EBSD** Select Data Files EBSD ctf W56-T0-P2.ctf Pole Figures EBSD ODF Tensor Plot << Previous Next >> Finish	 Import Wizard **Crystal Reference Frame for Phase 0** Crystal Symmetry Mineral: Indexed [✓ Not Indexed] mineral name: notIndexed plotting color: ▨ Crystal Coordinate System Point Group: 1 Axis Length a b c Axis Angle alpha beta gamma	 **Crystal Reference Frame for Phase 1** Crystal Symmetry Mineral: [✓ Indexed] Not Indexed mineral name: Magnesium plotting color: 🟦 Crystal Coordinate System Point Group: 6/mmm, X||a*, Y||b Axis Length a 3.209 b 3.209 c 5.21 Axis Angle alpha 90 beta 90 gamma 120
Figure 2: Dialog for selecting the file	*Figure 3: First material (unindexed points)*	*Figure 4: Next (and last material)*

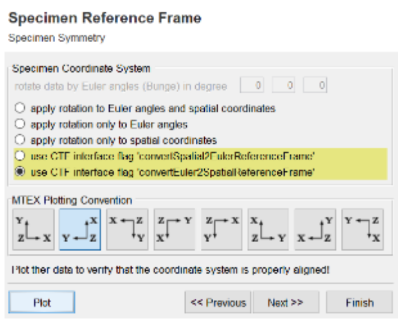
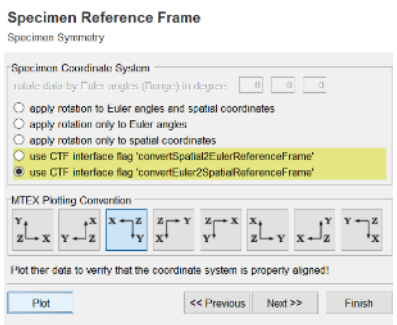
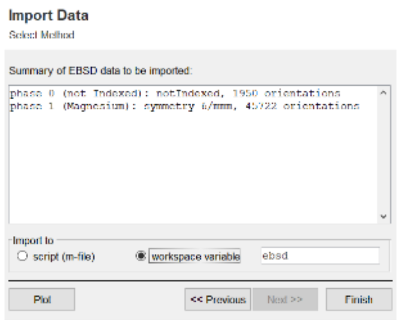
 **Specimen Reference Frame**
Specimen Symmetry
Specimen Coordinate System
rotate data by Euler angles (Bunge) in degree 0 0 0
○ apply rotation to Euler angles and spatial coordinates
○ apply rotation only to Euler angles
○ apply rotation only to spatial coordinates
○ use CTF interface flag 'convertSpatial2EulerReferenceFrame'
● use CTF interface flag 'convertEuler2SpatialReferenceFrame'
MTEX Plotting Convention
Plot ther data to verify that the coordinate system is properly aligned!
	 **Specimen Reference Frame**
	Specimen Symmetry
	Specimen Coordinate System
	rotate data by Euler angles (Bunge) in degree 0 0 0
	○ apply rotation to Euler angles and spatial coordinates
	○ apply rotation only to Euler angles
	○ apply rotation only to spatial coordinates
	○ use CTF interface flag 'convertSpatial2EulerReferenceFrame'
	● use CTF interface flag 'convertEuler2SpatialReferenceFrame'
	MTEX Plotting Convention
	Plot ther data to verify that the coordinate system is properly aligned!
		 **Import Data**
		Select Method
		Summary of EBSD data to be imported:
		`phase 0 (not Indexed): notIndexed, 1950 orientations`
		`phase 1 (Magnesium): symmetry 6/mmm, 45722 orientations`
		Import to: script (m-file) [✓ workspace variable] ebsd

Figure 5: Orienting your data. This setting gave the plot in Figure 9, which is wrong.	*Figure 6: Selecting the MTEX plotting convention of X to the west gives the correct spatial plot, shown as Figure 10.*	*Figure 7: Importing the data as a variable named `ebsd`.*

Validate your settings: a checklist

It would be great if there was some kind of universal convention for how orientation of EBSD data should be recorded. Unfortunately, that doesn't exist. Therefore, MTEX has no idea which way around your map should be plotted, and you must tell it.

It's also possible that the acquisition software uses one set of XY axes to plot your data spatially (i.e. map coordinates) and a different set of XY axes for the orientation information. Yes, this would mean you have two different coordinate systems for the same data. Yes, I think it would be a bad idea to do this… But Brucker did this for the first revision of their software, and then *changed how data was saved in later versions without flagging this for users*. So even if your data was imported and validated in the past, the equipment or equipment settings can change.

Each time you acquire a map, verify that at least #1 and #2 below are correct. For each new instrument, I strongly recommend checking #3 and 4 the first time you use it. #5 only becomes important if you're working with scans taken at a low magnification.

1. Validate your XY map spatial layout (see)
2. Validate your EBSD Euler orientations (see)
3. Validate your crystallographic convention (three options below)
    - Check the crystal symmetry in the EBSD variable (`ebsd.CS`)
    - Plot the unit cell with axes shown (Figure 11)
    - Plot the pole figure of the unit cell (Figure 17)
4. Validate your spatial measurements
5. Validate your angular measurements

Also remember that images on your EBSD acquisition system display can be oriented differently from the images on your SEM (vertical flipping is especially common).

Validate your XY map orientations

Save an image of a map (say IPF in x) as shown on your data acquisition equipment (e.g. your EBSD detector software) during acquisition and compare it to a map plotted with MTEX. You can generate a map in MTEX by clicking the 'plot' button during the import process (See Figure 6) or by using the commands below.

This code assumes you've imported your data. If not, try it out by typing `mtexdata twins` to load some example data. Note that you need to change `Magnesium` below to your phase of interest.

```
%create a color map of the IPF orientations of the selected material
oM = ipfHSVKey(ebsd('Magnesium'));
%define the direction of the ipf
oM.inversePoleFigureDirection = xvector;
%convert the ebsd map orientations to a color based on the IPF
color = oM.orientation2color(ebsd('Magnesium').orientations);
figure; plot(ebsd('Magnesium'),color);
```

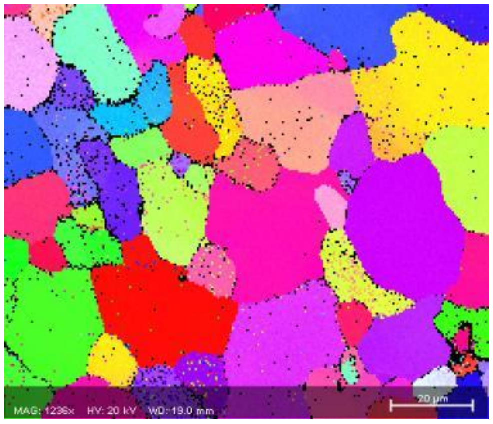
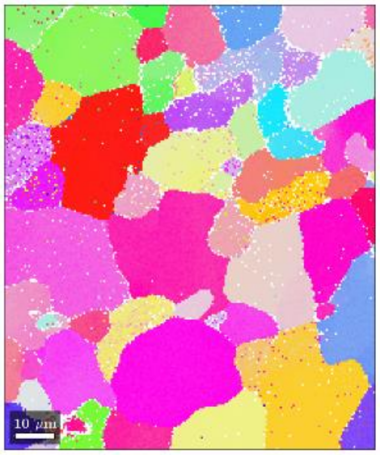
	

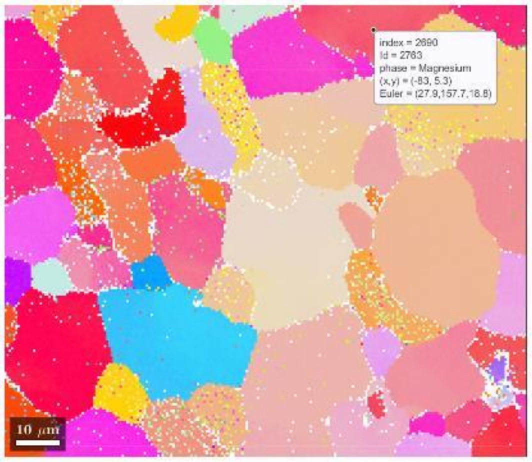
Figure 8: IPFX map as shown by EBSD system during acquisition by Bruker software	Figure 9: IPF map as plotted by MTEX. Note that the spatial orientation is off by 90°, so you must adjust the import settings.

When the layout looks the same between your data acquisition system and MTEX, you've validated the spatial orientation of your data. Comparing Figure 8 to Figure 9, you can see that the map is not yet oriented correctly. Now compare Figure 8 to Figure 10, although the colours are different the grain layout is the same.
If you click on the map, a tooltip will tell you the XY coordinates. Figure out which direction is positive X, and which is positive Y. You can customize the import settings to make this your default (see) but be sure to check your map was imported correctly each time by comparing it the data directly from the acquisition system. Note that each time you install or update MTEX you will need to set all defaults again.

Validate your Euler orientations

In your EBSD acquisition software (i.e. while you're acquiring data), spot check the Euler angles of a couple different points from different grains and either capture this image or make a sketch in your notes including the Euler angles. Once the map is plotted in MTEX, click on the same grain and the Euler angles of that point will pop up (see Figure 10). If the Euler angles are not identical, you've made an error in importing (typically things are out by a multiple of 90°). Delete any incorrect imported ebsd variable, change your import settings, and try again. Alternatively, rotate the orientations of the data as discussed in by multiples of 90° until Euler angles in MTEX match those of the original acquisition system within a degree or so.

If you're out by some value not equal to 90°, check you didn't make a mistake, then talk to whoever maintains the system.



```
index = 2690
Id = 2763
phase = Magnesium
(x,y) = (-83, 5.3)
Euler = (27.9,157.7,18.8)
```

Figure 10: Checking Euler orientations by clicking on the map

Crystallographic convention validation

Even if the Euler angle values and map orientation are both correct, the system may be using a different relationship between the crystallographic unit cell and the sample axes (see) In other words, **two points can have the same Euler angle but create different pole figures due to a difference in how the relationship between crystallographic unit cell and sample axes was defined.** Three methods for validating your settings for this are included at the start of section .

Therefore, we must define how MTEX should relate each crystallographic unit cell to a set of Cartesian XYZ axes. This set of orthogonal x,y,z axes provide a consistent reference between the crystallographic unit cell and the rest of the sample. For a cubic cell, the crystal axes are aligned with the corners of the unit cell and everything is very straightforward. However, for non-cubic unit cells there is no universal convention for how the orthogonal x-y-z axes will line up with the non-orthogonal crystallographic unit cell.

MTEX has a default for each type of unit cell, which may or may not be what you want. Different journal article authors discussing non-cubic crystallography will often have their own convention which may or may not be stated explicitly in the article.

In MTEX, for each phase the unit cell information is included in the `ebsd` variable and can be seen by typing `ebsd.CS` or `ebsd('magnesium').CS` and hitting enter. The symmetry (in Hermann-Mauguin short notation) and relative unit cell axis lengths will be listed, and for non-cubic systems a reference frame (the relationship between the crystallographic unit cell and reference orthogonal system) is also listed.

Crystal directions in HCP unit cells

For HCP unit cells, the default relationship is defined as z parallel to the c axis, y//to [-1,2,-1,0] and x//to [1,0,-1,0]. This is shown as Figure 11, and the code to create this figure is as follows (assuming you have HCP data loaded in a variable named `ebsd`).

```
%copy the crystal symmetry from the ebsd variable.  This variable provides
the relative axes of the unit cell (c/a ratio)
cs=ebsd.CS
%using the crystal symmetry customize the HCP unit cell drawn by the crystal
shape command.
cShape=crystalShape.hex(cs)
%plot the shape
plot(cShape,'figSize','small')
%hold the axes, so the new plot is on top of the old
hold on
%add arrows to the figure along the principal axes, scaled down
arrow3d(0.5*[xvector,yvector,zvector],'labeled')
hold off
%manually set the viewing angle of the plot
set(gca, 'CameraViewAngleMode', 'manual', 'CameraTargetMode', 'manual',
'CameraPositionMode', 'manual');
```

If you want MTEX to use a different relationship in order to have your analysis match a literature example, you can manually enter a unit cell with the following syntax, and the effect of this change is shown as Figure 12.

```
csx2a=crystalSymmetry('6/mmm',[3.2,3.2,5.2],'X||a','Z||c')
```

You cannot edit the crystal symmetry of your EBSD variable directly. If you want to change your EBSD data to align with a more complex reference system, you want the `transformReferenceFrame` command (see documentation online), which changes the crystal symmetry, and simultaneously updates all Euler angles to compensate.

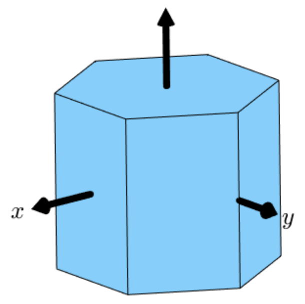
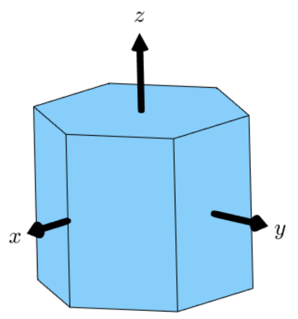
	

Figure 11: Plot of an HCP unit cell, showing the MTEX default axes	*Figure 12: Plot of an HCP unit cell, with axes updated, no longer the default*

Spatial validation

If your system was calibrated properly, the EBSD system scale should precisely match that of your SEM system, which should be accurate. However, following EBSD measurements on a magnesium sample I lightly etched it in Nitol. The carbon contamination from the EBSD scan formed a perfect etch resist, and I could see my scan areas very precisely (see Figure 13). This is a very useful technique and will tell you if the scan got significantly distorted (usually due to tilting issues or charging) so that you can crop off the damaged part or compensate.

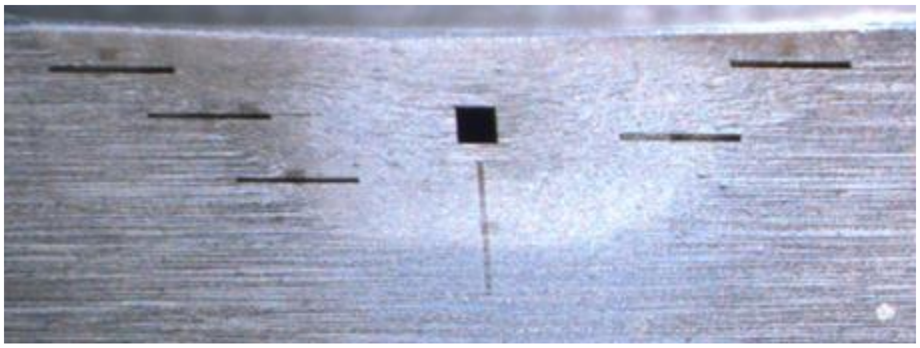


Figure 13: Light etch with 2% nitol following EBSD scans reveals the locations perfectly on this AZ80 magnesium friction stir weld. Horizontal EBSD line scans were 1000 µm according to the EBSD acquisition system but were actually 1100 to 1200 µm.

The size of the scanned areas did not match. Either my optical microscope calibration was incorrect, or that of the EBSD system was. It turned out to be the latter- our EBSD system was off by 11% in terms of spatial measurements (note that it's also possible for the EBSD system to be wrong differently in X and Y due to tilt correction issues). Therefore, I recommend verifying your system's spatial measurements which can be done with a silicon grid in your SEM and verified periodically. Or you can use the etch-resist method shown in Figure 13.

If you acquired data and then found it has an incorrect spatial scale, it's easy to correct. The easiest method is to fix this after MTEX import, as described in the section . Alternatively, you can edit the .ctf file manually in a text editor, spreadsheet, or MTEX. Note that if you edit the .ctf manually, both the file header and the xy coordinates of each point must be updated, the latter of which can be done with a script.

Parallelogram scans: when things go wrong.

The nice thing about MTEX is that it can deal with odd cases like the one shown in *Figure 14* and *Figure 15*. To fix this, once the data is imported and you've made precise measurements of the actual area scanned from an etch resist image of the area (e.g. *Figure 14*) you do an on the EBSD variable's x and y coordinates so that your data matches the actual scanned shape. Your map in MTEX will now be a parallelogram, reflecting reality.

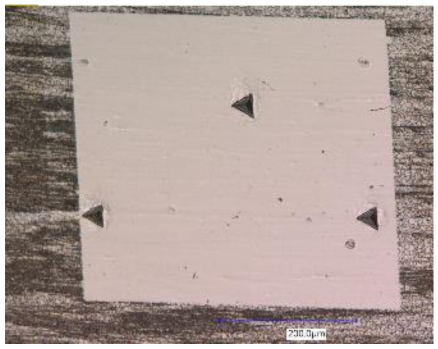
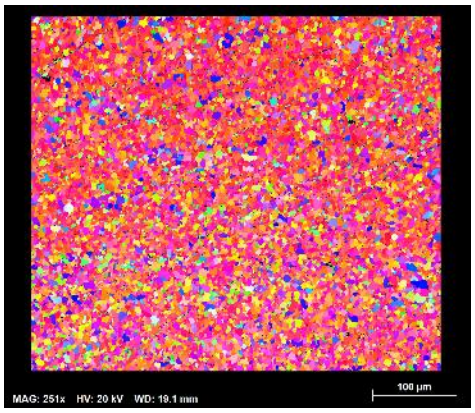
	

Figure 14: The post-scan area of a distorted scan (the indents were made post-scan). Note the distorted scan area	*Figure 15: The resulting map. Note the lack of visible distortion. As far as the EBSD system is concerned, everything was fine.*

Angular validation

When creating lower magnification EBSD scans, the distance between the sample and the detector is different enough between the center and edges of the scan that it can induce errors. Despite calibration routines that should compensate for this, it can persist. Figure 16 shows a low magnification EBSD scan across a silicon wafer, which shows about a 5 ° difference between the center of the scanned area and the edges. This is an artifact, but difficult to compensate for. If you're doing small scans (~ 100 um across) or don't care about this kind of deviation, you can just ignore this effect, but it's worth getting an idea of how good your system is in this respect. A small chip of silicon wafer can be easily attached to another sample and scanned quickly – tight spacing of scanned points is not required.

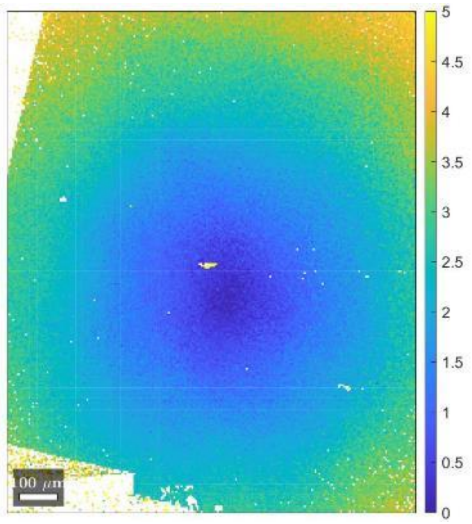


Figure 16: Angular error of EBSD measurement across a silicon wafer (scale in degrees)

Saving progress, outputting data, and saving figures

Once you've gone to the trouble of importing and cleaning up your EBSD data you won't want to do it twice. Save the imported data as follows.

Table 1: Some basic MATLAB commands for saving and exporting

Command	Description
`save`	Saves a file with the name matlab.mat containing all workspace variables in the current directory. **Avoid this as it's easy to overwrite by accident**.
`save('foo')`	Saves a file with the name foo.mat containing all workspace variables in the current directory. **This is the correct way to save your data.**
`cd .. or cd('filename')`	Changes the working directory up one level or to a filename
`print('-dtiff','-r500','IPFY')`	Save current figure to the current working directory as a tiff with a resolution of 500 and file name IPFY.tif
`xlswrite('Foo.xls',variable,'SheetName')`	Export a variable to an .xls file named Foo.
`strmin = ['Neighbours',num2str(NborNum)]; save(strmin,'StrAccom');`	To save variable `StrAccom` to the current directory with a filename that is a mix of text and variable (e.g. `Neighbours8`).

Table 2: Some useful MATLAB commands for working with figures

`figure`	Creates a new active figure. MTEX applies all plot commands to the active figure by default.

`hold on`
`hold off`
	Forces plotting on top of an existing figure. Otherwise, MTEX erases the previous plot.

`close all`	Closes all figures
`clear all`	clears all workspace variables

Table 3: Some useful MTEX functions

[val,idnum]=max(ebsd.x) [BigSize,bigGrainID]=max(grains.grainSize)	Returns the maximum value of the parameter, and the associated ID. Parameter must be an integer.

Some general notes on MTEX

Both MTEX and MATLAB are case sensitive. Note curved apostrophes are not accepted, and – is always a minus sign and cannot be used in variable names. A semicolon at the end of a command stops displaying the output, so if you're trying to understand what the code is doing removing these can help.

A good way to work through scripts line by line is to copy and paste them into the script editor (hit the plus for a new tab) highlight the line in question and hit 'F9' on your keyboard (doesn't work on some laptops).

Internally MTEX works in radians. To convert radians to degrees, use the MATLAB command rad2deg(variable) or *degree. For example, you commonly input values in degrees, so to tell MTEX to use 10° as a value, use `10*degree`. In contrast, MTEX commonly outputs values in radians, so if you want your plot to have a scale in degrees instead of radians, you would use the syntax `/degree`. To convert a value from radians to degrees as a stand-alone command use `rad2deg(variable)` or `/degree`.

Test data sets/ sample data

MTEX has sample datasets for you to experiment with. Find a list of these by typing `mtexdata` and then enter, but note that some of them are pole figure inputs (not covered in this manual). The `forsterite` dataset is particularly good for working with multiple phases, while the `twins` dataset is good for magnesium. To load a dataset just use a command like: `mtexdata forsterite.` When you load a data set and you get an error message about saving, it may be trying to save to a read only folder (where MTEX was installed). Opening MATLAB as an administrator or changing the current directory fixes this.

Targeting MTEX functions appropriately

Specimen directions, crystallographic indexes, and orientations

MTEX commands deal with both spatial positions and angles of all kinds. These include references to crystallographic planes and vectors (with Miller Indices), orientations (with Euler angles), and 3D vectors (with Cartesian vectors). If you don't clearly understand the differences between the three, and which one you need for a given command, you're going to have a lot of errors and frustration.

Specimen Directions (vector3d)

A vector3d is **defined relative to the defined x,y,z orthogonal axes of the imported sample spatial data**. In other words, vector3d are defined relative to your EBSD map as a system of x, y, z values, where x, y, and z are all perpendicular to each other. This is your classic Cartesian system, and defining x and y will also define z via the right-hand rule.

Three different ways to define a vector3d using x,y,z values:

```
%define the variable v as a vector in the +x,+y direction (bisecting the two
axes)
v = vector3d(1,1,0)

%define r as a vector parallel to the positive x axis of the map.
r= xvector

%define c as a vector3d using combinations of the x and y axis.
c = -xvector + 2*yvector;
```

Or define a vector3d using spherical coordinates:

```
%convert spherical coordinates to a Cartesian vector.
polarAngle = 60*degree;
azimuthAngle = 45*degree;
%use these angles to create a vector3D.  Note the first angle is the angle to
the z axis, while the second is the angle to the x axis.
v = vector3d.byPolar(polarAngle,azimuthAngle)
```

Overlay a vector3d on an existing plot:

```
hold on
arrow3d(0.5*[xvector,yvector,zvector],'labeled')
```

Crystal Directions (Miller indices)

A Miller index **defines the orientation of a plane or vector within the crystallographic unit cell** (cubic, HCP etc.), and thus each Miller index definition needs the applicable crystal symmetry/ unit cell information as input. Miller indexes are based on the crystallographic axes of the unit cell, which may not be Cartesian axes.

Consequently, the crystallographic unit cell axes are named a b c, to help distinguish from the sample directions x y z. For example, the HCP unit cell is commonly represented by two equal-length ‘a’ axes 120° apart and a ‘c’ axis perpendicular to them. BCC or FCC unit cells will typically have three orthogonal axes that are all equal length, but other cubic system axes may be orthogonal and different lengths. Unit cell proportions do not affect the Miller indexes, but are used for other functions, (see), and internally for calculations.

Miller indexes can reference a vector or a plane in the unit cell. Planes are defined by using the plane normal (which is a vector). On plot annotations, MTEX distinguishes vector and plane Miller indexes by using labels of () for one plane or {} for the plane family, while vectors are indicated by [] for a single direction or <> for the vector family. This is a common convention in materials. If you don't include the appropriate tag when you define the Miller index, the system assumes you're referring to a plane normal.

Sample code for inputting HCP Miller indexes by four methods

In a cubic system, there's no difference between a vector and the plane normal with the same index, but in HCP materials the normal to a plane is not necessarily parallel to a vector with the same indexes (see Figure 17 for a demonstration of this).

In HCP systems, both the Miller and Miller-Bravais indexes are commonly used. Combined with planes and vectors, this gives you four ways to define an index. To review, Miller indexes have three indexes, while Miller-Bravais have four (the first three values sum to zero). The Miller-Bravais system is a bit more intuitive, as equivalent planes or directions in the HCP unit cell have values that have the first three numbers as permutations of each other, and so I use it by default. For terminology, see the table below- you will need to use this tag to specify what type of index you want to define. Note that these arguments (unlike most of MTEX) are not case sensitive as of version 5.5.2. If you get an error, check your version, or try the other case.

Table 4: Tags for Miller and Miller Bravais definition.

	Direction/vector	Plane
Miller Index	uvw	hkl
Miller Bravais index	UVTW	HKIL

Four examples defining a miller index in an HCP system. Note that converting between the Miller and Miller-Bravais versions for a plane is straightforward (just add the missing third-position value such that the first three values sum to zero), while for a direction you need a more complex equation, look it up online. If unspecified, the system assumes the Miller index defines a plane.

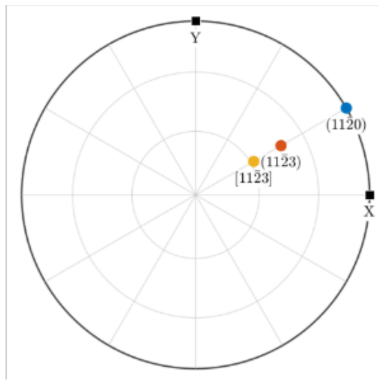
```
%load some ebsd data if necessary
mtexdata twins
%define the crystal symmetry to use, pulling it from the ebsd variable.
cs=ebsd('Magnesium').CS;
m1=Miller(1,1,-2,3,cs,'UVTW') %Miller-Bravais index definition of a direction
m2=Miller(1,1,3,cs,'uvw') %Miller index definition of a direction
m3=Miller(1,1,-2,3,cs,'HKIL') %Miller-Bravais index definition of a plane
normal
m4=Miller(1,1,3,cs,'HKL') % Miller index definition of a plane normal
```

Plotting Miller indexes on pole figures

Miller indexes can be plotted on a pole figure just like you would a set of Euler orientations. In this case, MTEX creates the pole figure while assuming the xyz axes of the unit cell are oriented identically to the xyz axes of the spatial data. In other words, the orientation of the point for which you plot Miller indexes is assumed to be Euler angle: (0,0,0). So, for an HCP system using the default unit cell definition, if you've got the map with x positive to the left, MTEX will orient the [1,0,-1,0] axis of an HCP unit cell facing to the left. Therefore, if you change how your spatial data is laid out **or** plotted, your pole figures of Miller indexes will also change.

The following example produces Figure 17 and provides an example of how MTEX plot Miller indexes of planes and directions. Note how the brackets of the labels change based on the type of Miller index (plane vs direction).

```
%load some test data
mtexdata twins
%copy the symmetry to a new variable
cs=ebsd('Magnesium').CS
%define the Miller indexes you want to plot
m0=Miller(1,1,-2,0,cs,'HKIL')
m1=Miller(1,1,-2,3,cs,'HKIL');
m2=Miller(1,1,-2,3,cs,'UVTW')
%create a figure
figure;plot(m0,'upper','labeled','backgroundColor','white','grid','MarkerSize
',10)
hold on
plot(m1,'upper','labeled','backgroundColor','white','grid','MarkerSize',10)
plot(m2,'upper','labeled','backgroundColor','white','grid','MarkerSize',10)
%annotate the x and y directions
annotate(xvector,'label',{'X'},'BackgroundColor','w')
annotate(yvector,'label',{'Y'},'BackgroundColor','w')
```



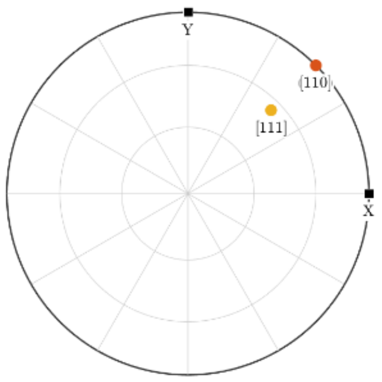


Figure 17: Plot of a plane normal and a direction in an HCP unit cell, showing they are not interchangeable. Note that this is a pole figure of the UNIT CELL so the c-axis (0001) is in the center and (11-20) is on the edge.

Figure 18: Plot of a plane normal and a direction in a cubic unit cell, showing they are the same (the dots overlap). Note that this is a pole figure of the UNIT CELL so the cubic axes <001> are in the center and the edge of the pole figure.

Miller indexes for a cubic system

In a BCC or FCC system where everything is perpendicular and all sides of the unit cell are the same length, matters are much more straightforward. There are only three-digit indexes, and you don't need to specify whether you're discussing a vector or the plane normal with the same indexes, since they're identical. Therefore, the code can be simplified as you don't need any tag. Here is an example that generates Figure 18.

If you're in a system where the sides of the unit cell are different lengths, you're back to needing a tag (`'hkl'` or `'uvw'` as appropriate, see Table 4), since the plane normal and the vector with the same index are no longer equivalent. If you omit the tag, the index defaults to being treated as a plane.

```
%load some test data
mtexdata copper
%copy the symmetry to a new variable
cs=ebsd('C').CS
%define the Miller index you want to plot
m0=Miller(1,1,0,cs,'hkl')
m1=Miller(1,1,0,cs,'uvw')
m2=Miller(0,0,1,cs,'uvw')
%create a figure
```

```
figure;plot(m0,'upper','labeled','backgroundColor','white','grid','Mar
kerSize',10)
hold on;
plot(m1,'upper','labeled','backgroundColor','white','grid','MarkerSize',10)
plot(m2,'upper','labeled','backgroundColor','white','grid','MarkerSize',10)
%annotate the x and y directions
annotate(xvector,'label',{'X'},'BackgroundColor','w')
annotate(yvector,'label',{'Y'},'BackgroundColor','w')
```

Orientations (Euler angles)

Internally MTEX stores orientations and makes any associated calculations as Quaternions, but typically we work with Euler angles instead. Bunge convention is the MTEX default for Euler angles and will be used here. An **orientation defines the minimum rotations around the X-Y-Z axes to convert a given EBSD point to align with the default unit cell lattice.** Since the unit cell xyz axes are aligned with the sample axes, orientations are affected by both your spatial layout and your unit cell definition.

In other words, MTEX takes the spatial data axes (vector3d) which are set by the user, defines a unit cell with a fixed relationship(crystal direction) relative to these sample spatial axes, and then calculates the rotations to convert each EBSD point to line up with the unit cell (orientation). This is an orientation. A misorientation is when one orientation is converted into another orientation (e.g. defining how one EBSD point should rotate to become equivalent to a second EBSD point).

Therefore, orientations describe how the points of the EBSD map are oriented with respect to the sample. To work in the opposite direction, we use the inverse of an orientation. Orientations may be plotted using the `plotPDF` command for pole figures, which are covered extensively in this manual starting at . Although not covered in this manual, orientations are also used with the `plotIPDF` command for inverse pole figures or the odf command. Let’s take a quick look at some pole figures, with the code below (see Figure 19 for output)

```
%load the MTEX test dataset for magnesium
mtexdata twins
%load the orientations of the ebsd map into a separate variable
o= ebsd('Magnesium').orientations
%load the crystal symmetry into a variable
cs= ebsd('Magnesium').CS
%define the pole figures you want to create.  Note that these indexes are
assumed to be plane normals by default
h=[Miller(0,0,0,1,cs,'hkil'),Miller(1,0,-1,0,cs,'hkil'),Miller(1,1,-
2,1,cs,'hkil'),Miller(1,1,-2,0,cs,'hkil')];
%plot the pole figure
plotPDF(o,h)
```

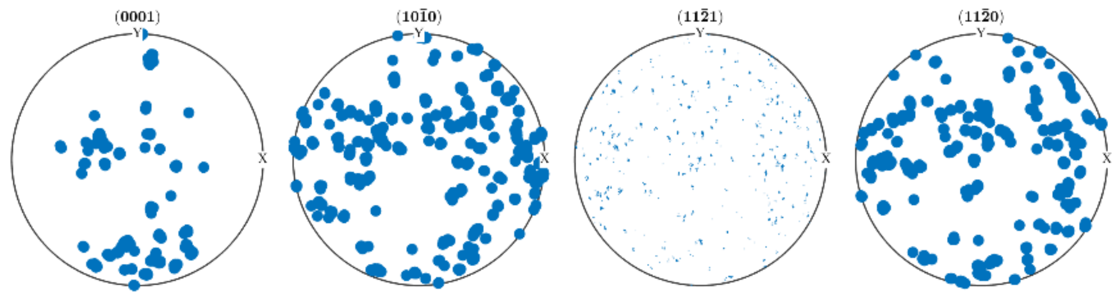


Figure 19: Pole figure of 416 random orientations from the `twins` *dataset.*

For *Figure 19* note that the data is reduced to 416 randomly selected orientations from the map (unless you force it to plot all), so if you plot the map twice you'll get something that looks slightly different each time. Plot another set of pole figures using the code below and compare them. The resulting pole figures will be similar, but not identical to the first set.

```
figure;plotPDF(o,h)
```

The size of the dots depends on how many points per figure (you can also control this; we'll discuss that later). So even though the same 416 orientations are plotted in each pole figure, a single orientation generates one point in the (0001) plot, three points in the (10-10) and (11-20) plots, and six points in the (11-21) plot. These are called 'reflections' because they're essentially identical copies of the same orientation and observed during diffraction experiments. Plot a single orientation from your data as follows:

```
%take the fifth point from the orientation set we defined, and move it to its
own variable
o1=o(5)
%plot just this single orientation.
figure;plotPDF(o1,h)
```

Orientations can be defined by many methods

```
%in terms of Euler angles
% zxz explicitly means Bunge convention used (this is the default).
o = orientation('euler',30*degree, 30*degree, 30*degree,'ZXZ',cs);
figure;plotPDF(o,h)
```

Define in terms of a rotational axis and an angle: this takes an Euler angle of 0,0,0 and rotates it around the axis by the angle

```
%define an axes based on a sample direction
v=vector3d(xvector);
%define a rotation
omega=30*degree;
%define your orientation
```

```
ori = orientation.byAxisAngle(v,omega,cs)
figure;plotPDF(ori,h)
```

Get an orientation by extracting xy coordinates by clicking on an EBSD map

```
%plot an ebsd map
plot(ebsd('Magnesium'),ebsd('Magnesium').orientations)
%get one point from the map
[x,y]=ginput(1);
%get the orientation based on xy coordinates.
ori1=ebsd(x,y).orientations
figure;plotPDF(ori1,h)
```

Misorientation calculations

Because misorientations are calculated by rotating one orientation into the sample frame of reference and back into another orientation, misorientations may also be represented by Euler angles. In essence, a misorientation is an orientation PLUS a reverse orientation.

Misorientation between two EBSD points is not the same as the minimum angle between them. A good way to think of it is that misorientation is a set of instructions for aligning unit cell A to with unit cell B, while an angle is typically defined between two directions in the unit cell e.g. between the (0001) axes of each unit cell. Further, a misorientation includes more information, it is represented as an Euler angle, and both the misorientation axis and angle can be extracted. For a pair of given orientations the angle is typically lower than or equal to the misorientation angle.

```
%plot an ebsd map
plot(ebsd('Magnesium'),ebsd('Magnesium').orientations)
%select two points from the EBSD map. If you pick a point in the twin and a
point in its parent you can get the relationship between them
[x,y]=ginput(2);
%get orientations from these xy coordinates
ori1=ebsd(x(1),y(1)).orientations
ori2=ebsd(x(2),y(2)).orientations
%calculate the misorientation moving from ori2 to ori1.  Note that this is an
operation where the order truly matters.
mori = inv(ori1) * ori2
%determine the axis of misorientation, rounded to integers
round(mori.axis)
%determine the misorientation in degrees
mori.angle/degree
%compare this to the angle between the (0001) axes
M=Miller(0,0,0,1,cs,'hkil');
%convert the orientations to vector3d
v1=ori1*M
v2=ori2*M
%calculate the angle between them, using the antipodal tag to use twofold
symmetry
angle(v1,v2,'antipodal')/degree
```

Relationships and converting between Orientations, Vector3d and Miller

You can convert an orientation (defined below) to a vector3D, and often need to for calculations. Below is one example.

```
%continuing on from the previous example, compare the misorientation that was
calculated to the angle between the (0001) axes
M=Miller(0,0,0,1,cs,'hkil');
%convert the orientations to vector3d
v1=ori1*M
v2=ori2*M
%calculate the angle between them, using the antipodal tag to use twofold
symmetry
angle(v1,v2,'antipodal')/degree
```

This code takes the orientation in the variable `ori1` and determines the c axis (defined in the Miller index) as a vector3d (i.e. relative to the specimen axes). This is useful for doing comparisons which only consider the 'c' axis.

You cannot convert from a vector3d to an orientation because there is insufficient information to orient a crystal (i.e. even orienting one crystallographic axis with respect to the specimen is not enough, you need more information to uniquely define an orientation).
Because there is a one to one ratio between vector3d and Miller indexes, you can convert from crystallographic directions to specimen directions and back again per Table 5). Note that this conversion is based on the relationship between the unit cell xyz axes and Miller indexes as discussed in)

Table 5: Table of conversions. Columns are starting data type; rows are resulting data type.

	Vector3D (v)	Miller (m)	Orientation (o)
Vector3D	■	v=vector3d(m)	v=o*m
Miller	m = Miller(v,cs)	■	
Orientation	Not possible	Not possible	■

Getting to know the EBSD variable

The `ebsd` variable you've loaded and worked with has a custom MTEX data type (named EBSD). Double clicking on the variable name in the workspace will let you see what's stored there. One thing you need to be aware of is that there's a hierarchy of data in this variable. The top level includes fields that apply to all phases of your EBSD data (see Table 6) while the lower level is phase specific (see Table 7).

There are a lot of fields, so I've only presented the most commonly used ones, in the context you're most likely to use them. To get a more complete list, type `ebsd.` into the command window and hit tab. Note that many fields have sub-fields as well.

Referencing data in the EBSD variable

Top level data is accessed by using `ebsd.`_____ to access all the entries/points simultaneously. To address a specific data point, one method is to use `ebsd(i).`_____ where `i` is an integer which represents the i-th position in the `ebsd` structure. Each data point also has an id number, assigned sequentially when imported. When the ebsd data is first imported, the point with `ebsd.id`=100 is the 100th entry in the `ebsd` variable and can be accessed by `ebsd(100)`, so the two methods are equivalent. However, if you delete data (such as non-indexed points) or make a subset of the original variable, the id number for a given EBSD point will no longer equal the sequential position in the `ebsd` variable. **Therefore, when addressing a specific EBSD point it's much better to use the point's ID, which is consistent and maintained when creating subsets or deleting data.** EBSD points only get a new ID if you use the `gridify` command on them. The `gridify` command changes the `ebsd` variable from a linear variable to a 2D matrix layout, and is only mentioned briefly in as it is not necessary for most analysis work.

To examine one EBSD point, you can use syntax such as the line below, where you first generate a logical of all points that have an id of 500 (since ID numbers are unique this is only one point) then use that logical to filter the original `ebsd` variable down to that point. Any MATLAB accepted logical can be used within the brackets, so ranges can be selected this way as well.

```
%separate out ebsd point with id=500
temp=ebsd(ebsd.id==500)
```

You can get the maximum and minimum values stored in any variable by using commands like `max(ebsd.id)` to interrogate them. Note that `max`, `min`, and `mean` commands only work for those `ebsd` properties that contain scalar values. In other words, `max`, `min`, and `mean` commands don't work with orientations, rotations, etc. To get statistics/histograms about orientations you'll need to generate an odf (orientation distribution function) which generates a continuous equation representing your data. This topic is covered in .

Table 6: Selected top level `ebsd` variable properties.

`Variable name`	What it contains	General usage
`ebsd.id`	The id of the EBSD point. Sequential from 1 to the variable end	Most robust method for referencing a point, or series of points. One entry per point.
`ebsd.rotations`	The orientations of all the indexed phases, stored as a rotation. Length equal to EBSD variable. Split into sub-variables for each section of the Euler angles (phi1, Phi, phi2) and other parameters.	Generally better to access orientations at the phase-specific level.
`ebsd.unitCell` and `scanUnit`	The grid spacing used for your EBSD map, based on defining four corners of an EBSD point 'pixel'	Used when adjusting the spatial scaling or position of your EBSD data.
`ebsd.phaseId`	The phase of the point, with 1 typically 'not indexed'. Length equal to EBSD variable.	Used to sort or limit your data or command by phase. One entry per point.
`ebsd.phase`	The phase of the point, with 0 typically 'not indexed'. Length equal to EBSD variable. Equals `phaseId` minus 1.	Also used to sort or limit your data or command by phase. One entry per point.
`ebsd.mineralList`	Cell array containing text name of the phase, index corresponding with phaseId	To correlate `phase` or `phaseId` fields with material names. One entry per phase, plus 'notIndexed'. Note that sometimes phases with no data points are listed here.
`ebsd.bc, ebsd.bands, ebsd.mad,ebad.bs` etc.	Quality measures of your EBSD data determined during acquisition. Which ones are included depends	Useful for removing outliers or other low-quality data. One entry per point.

	on your acquisition software and settings.	
`ebsd.x or .y`	The xy coordinates for each data point. Length equal to EBSD variable.	Necessary for spatial sectioning, reconnecting, scaling and rotation operations.
When you calculate grains, a `grains` variable is made, a sequential grain ID number is created for each grain, and the `ebsd` variable gains new fields. Typically, the properties below are added to the `ebsd` variable. Grain calculation is discussed in and uses syntax like `[grains,ebsd.grainId,ebsd.mis2mean] = calcGrains(ebsd('indexed'),'angle',10*degree);`		
`ebsd.grainId`	The id of the grain to which each EBSD point is assigned. Cross-references with `grains.id`	Useful for all operations on grains or groups of grains.
`ebsd('Magnesium').mis2mean`	Misorientation between each EBSD point and the mean value (e.g. average orientation) of that grain	Useful for misorientation plots

Accessing the lower level of the `ebsd` variable requires you to use the phase name of interest in your command (or the start of the name). Top level info can be accessed from the phase-specific (lower) level, but the reverse is not true. In recent versions of MTEX, many fields that were previously stored at the phase-specific level have been moved upwards.

Table 7: Phase-Specific `ebsd` properties.

`ebsd('Magnesium').orientations`
`ebsd('M').orientations`
`ebsd('m').orientations`
	The Euler angles of the phase specified.	Used for many EBSD maps and pole figures.

`ebsd('Magnesium').CS`	Provides information on the phase crystal symmetry.	Required for defining all crystal orientations and multiple other uses, including crystal shape.

Gridifying your data

When loaded, EBSD data generally continues to be in a linear array. In other words, if you type in `ebsd.id` you'll see that the id numbers start and 1 and continue to the end of the `ebsd` variable. The same for the spatial coordinates held in `ebsd.x` and `ebsd.y` However, there are a number of functions that are easier computationally if MATLAB's array functions are taken advantage of and the data is stored as a two dimensional array. In this case, the variable type is changed from EBSD to EBSDsquare or EBSDhex. The syntax for this command is `[ebsdGrid,newId] = gridify(ebsd)` and the gridified data is now mostly in 2D arrays of EBSD data, mimicking the size and shape of the original data map and re-numbering all the EBSD point ids.

Gridifying your data is a prerequisite for gradient, curvature and GND calculations, and makes large maps plot faster. However, we aren't discussing those functions in this doc (since they're fairly advanced) so just be aware that this is an option available to you.

Creating subsets based on `ebsd` variable parameters

All of the parameters listed in Table 6 and Table 7 can be used to filter your data or sort it into subsets. This will be covered again with examples specifically for grains later, but the process and syntax are identical. First you create a logical array (a set of 1 and 0 representing true and false respectively). This array will be equal in length to your `ebsd` variable. The logical array is then used to filter the `ebsd` variable, and only points that are 'true' are passed to the new `ebsd` variable that will be shorter in length. The first code snip below restricts the filtering to one phase (working with phase-specific commands), while the second one is a top-level command and affects all the `ebsd` data. Working with spatial subsets (extracting a rectangle, polygon, circle etc. from your map) is covered in .

```
%load the forsterite test data.
mtexdata forsterite;
%create a logical of all forsterite ebsd points with high band contrast
log=ebsd('Fo').bc>100;
%OPTIONAL determine how many points 'pass' this filter
sum(log)
%OPTIONAL determine how many points 'fail' this filter
size(ebsd,1)-sum(log)
%Create a new variable containing only the forsterite data.  If you don't do
this, the logical and the ebsd variable will be different lengths, and the
command fails.
filteredEBSD=ebsd('f')
%create the subset by filtering the forsterite ebsd data with the logical.
filteredEBSD=filteredEBSD(log)
```

```
%If you want to work with all phases simultaneously and not limit your result
to the forsterite data, it's even simpler
log=ebsd.bc>100;
filteredEBSD=ebsd(log)

%you can do the two above commands all in one step
filteredEBSD=ebsd(ebsd.bc>100)
```

When would I filter data based on the `ebsd` variable parameters?

One case where this is useful is to determine if a given feature you see on your map is based on high or low quality EBSD information. Keep in mind this judgment is based on using quality parameters recorded by your EBSD acquisition system and passed to MTEX when you loaded your data as a .ctf (or other) file. Depending on your EBSD acquisition system and settings, these measures may include band contrast (.bc), band slope (.bs), bands (.bands) mean angular deviation (.mad) or others. These measures of data quality will vary in how useful they are as an indicator of data quality.

In general, I find that band contrast plots (syntax below) are useful for locating physical damage that was not removed prior to the EBSD scan, such as scratches. Data from these regions is less reliable.

```
plot(ebsd,ebsd.bc)
```

An even more common case for filtering your data is where you want to isolate one phase (example syntax below).

```
forsteriteEBSD=ebsd('f')
```

Plotting Histograms of EBSD data.

Generating statistics on your EBSD data can be a useful visualization of your data. To create histograms of grain or orientation data see . Remember, you can only plot histograms of **numeric** values – not orientations or rotations or Miller indexes etc.

When would I plot a histogram using the `ebsd` variable?

This is particularly useful in deciding what parameters to use to filter your data. For example, load the `mtexdata small` dataset and determine how to get the highest quality data as measured by the band contrast parameter. What values should you filter with, and how much data will be left if you make that choice? You can easily get the max and min band contrast, but how are the points in between distributed? Below is an example of determining how much data will be eliminated at each threshold; in this case, it is evident that setting the filter to remove data below a band contrast value of 80 will remove about 49 points of the 1952, or < 3%. You could also determine this value with the command `sum(forsteriteEBSD.bc<80)` but often a visual representation is better.

```
%load the test data
```

```
mtexdata small
%extract the forsterite phase only
forsteriteEBSD=ebsd('f')
%get the max and min band contrast - turns out to range between 55 and 156
max(forsteriteEBSD.bc)
min(forsteriteEBSD.bc)

%plot a histogram dividing the data into four bins, using this max and min
%first, extract the band contrast values you want to plot
dataSet=forsteriteEBSD.bc
%define the histogram max and min
userMax=max(forsteriteEBSD.bc);
userMin=min(forsteriteEBSD.bc);
%divide the data into four equal bins
binNum=4;
%calculate the width the bins
width=(userMax-userMin)/(binNum);
%plot the histogram with specified bin ranges, and give it a handle.
figure;h1=histogram(dataSet,[userMin:width:userMax]);
%get information about the number of counts in each bin.
h1.BinCounts
%label your axes
xlabel('Band Contrast bin');ylabel('Counts')

%determine the number of points eliminated if you keep data with a band
contrast over 80
sum(forsteriteEBSD.bc<80)
%calculate this as a percentage of all forsterite data points
100*sum(forsteriteEBSD.bc<80)/size(forsteriteEBSD,1)
```

Working with EBSD Maps

Before we begin working with EBSD maps, you should understand the structure of an MTEX plot command.

```
plot(ebsd('Magnesium'),ebsd('Magnesium').orientations)
```

The first term inside the bracket provides the x-y spatial coordinates of the dataset to plot, and the second term dictates what the colour scaling is. Both the input for XY and for colour must be the same length. Colour scaling can be based on numerical values with a linear legend (see Figure 20), or orientations can be converted using a colour map (see Figure 21). See and Figure 36 for more details on controlling colour options and maps.

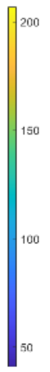
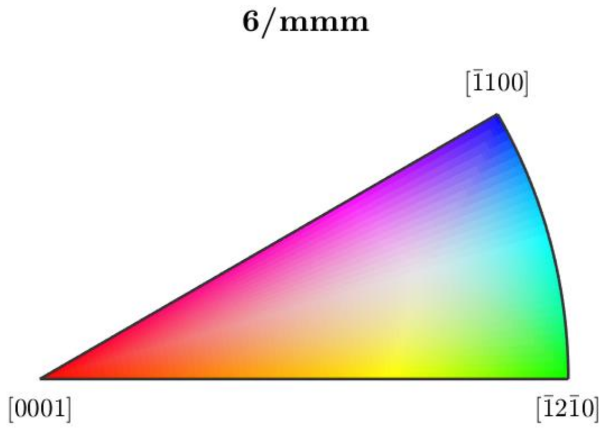
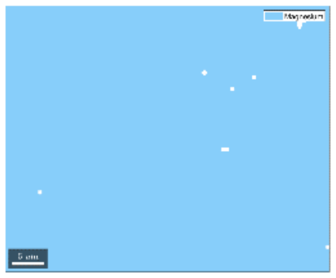
	

Figure 20: The default linear colour scale. There are many different scales for numerical data	*Figure 21: An orientation map for magnesium data. Plotted with* `ipfKey = ipfColorKey(ebsd('Magnesium').orientations)` `plot(ipfKey)`

You can skip the second term (for colour), creating the minimum code for plotting an EBSD map: `plot(ebsd('Magnesium'))` or `plot(ebsd('indexed'))` However, what you get from this code is only a map of where that phase exists in xy coordinates. For the `twins` dataset (which only includes one phase) you just get a blue square with holes indicating non-indexed points. Not very useful. For a multi-phase dataset such as forsterite, this is more useful and shows you where each phase is located spatially. Both of these are shown in Figure 22.



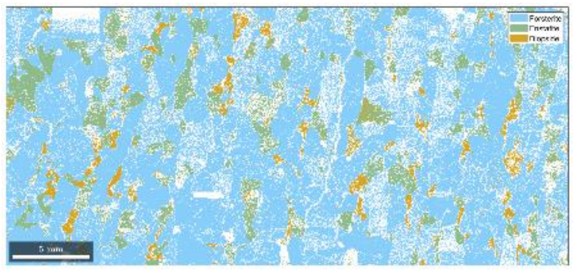


Figure 22: A map of all phases created with `plot(ebsd('indexed'))`. *Left,* `twins` *dataset, right* `forsterite` *dataset.*

Other parts of the `ebsd` variable can easily be called for plotting as well, as shown in Figure 23. Note that one limitation is that you can only plot multiple phases with one command if the colour scale is a numeric value.

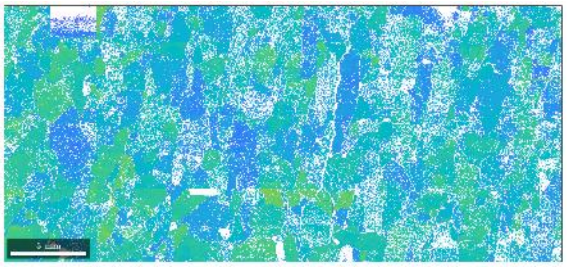


Figure 23: A plot of band contrast for all three phases of the `forsterite` dataset. Created with `plot(ebsd('indexed'),ebsd('indexed').bc)`

EBSD orientation maps

If you want to plot orientations, this uses a special orientation mapping function (see Figure 21) to convert orientation values to colours, and a different map is used for each crystal symmetry. So for the forsterite data set and other multi-phase EBSD data like it you need to plot one phase at a time so they can each be converted from a set of Euler angles to a colour value with the appropriate colour map. Sample code is below, and an orientation map generated from this is shown as Figure 24.

```
%load the forsterite test dataset
mtexdata forsterite
```
~~`plot(ebsd('indexed'),ebsd('indexed').orientations)`~~

The above plot command won't work, since there are multiple orientation maps called simultaneously. You get an error message

Instead, use:

```
%create a figure and plot the forsterite phase.
figure;plot(ebsd('f'),ebsd('f').orientations)
%cause the next two plots to display on top of the first.
hold on;
Plot the enstatite phase
plot(ebsd('e'),ebsd('e').orientations)
%plot the diopside phase
plot(ebsd('d'),ebsd('d').orientations)
hold off
```

If you only have one phase, things are easier, and `plot(ebsd('indexed'),ebsd('indexed').orientations)` works fine – try it with the `twins` dataset.

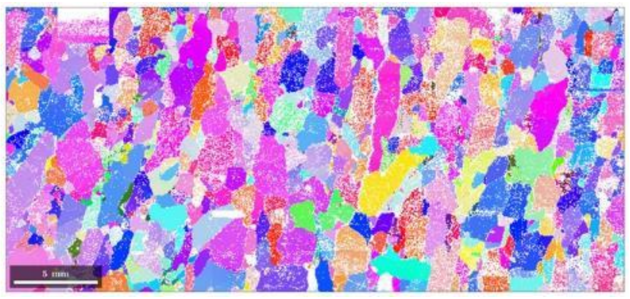


Figure 24: An orientation map showing three phases

Cleaning up your EBSD data or map

Once your data is loaded, you should plot a quick map (use the code in). Confirm that it is oriented to match the spatial orientation you saw during acquisition (see Figure 8 and Figure 9), check the Euler angles are correct (see Figure 10), and look for any of the other problems listed below.

- Incorrect spatial orientations or Euler angles (see)
- Incorrect scale in X-Y (see)
- Distorted map (see)
- Excessive data points your map (see)
- Crop or stitch your map (see
- Grid line issues (see)
- Remove noise (see)

Rotation of data

In some cases, your scan was not ideally aligned to the sample axis by a few degrees, or you want to rotate it for other reasons. The following methods will rotate the spatial data or Euler orientation in different ways. In all cases, you can either overwrite your EBSD variable, or make a new rotated EBSD variable and keep the original (this is usually the better method). If you have multiple `ebsd` variables, make sure you're calling the correct one when plotting a pole figure or doing other operations. MTEX uses the right-hand rule, so for the examples below since x is positive to the LEFT and Y is positive DOWN Z must be positive towards the viewer. **Note that in MTEX the spatial and EBSD orientations use the same axes.**

`rotate vs rotation`

`rotation` **defines** a rotation using an Euler angle, or by one of a dozen other definition methods.
The `rotate` command **reorients** your data (or vector or other object).
If you're just changing data in-plane (i.e. rotating about the z axis) then you can do everything with just the `rotate` command (see Table 8) but if you're doing more complex work or compound changes, you need to define a `rotation` first, and then call it in the `rotate` command.

Basic map Rotating and flipping

These commands rotate your EBSD data in plane. Keep in mind these commands can affect your spatial layout, Euler angles, or both.

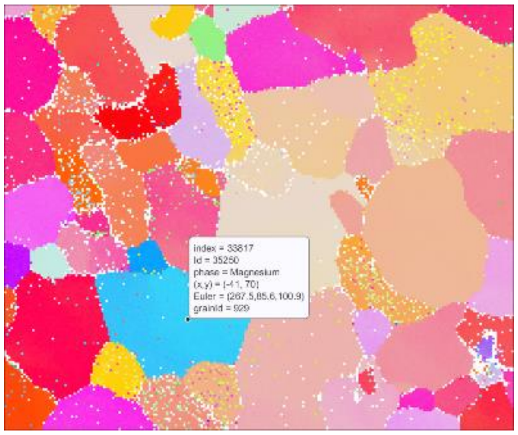
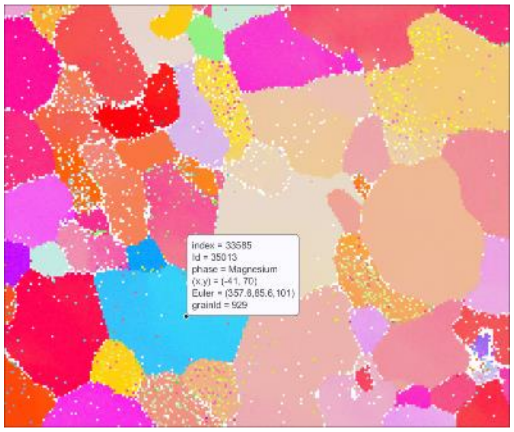
 index = 33817
Id = 35250
phase = Magnesium
(x,y) = (-41, 70)
Euler = (267.5,85.6,100.9)
grainId = 929
	 index = 33585
	Id = 35013
	phase = Magnesium
	(x,y) = (-41, 70)
	Euler = (357.8,85.6,101)
	grainId = 929

Figure 25: The starting image. Selected point has x=-41, y=70 and Euler =(267,86,101)	*Figure 26:* `ebsd2` *(see Table 8 for commands)* *Selected point has x=-41, y=70 and Euler =(**357**,86,101)*

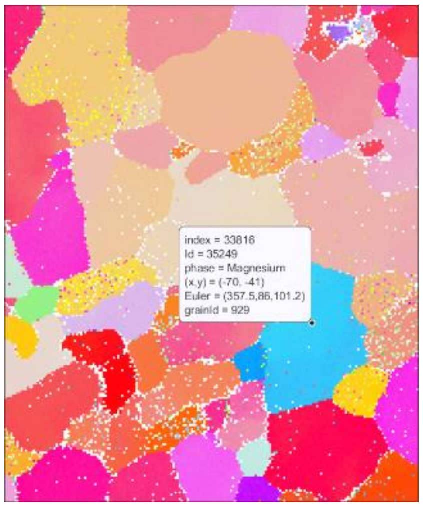
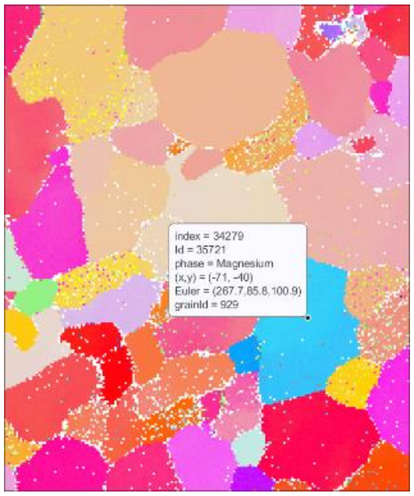
 index = 33816
Id = 35249
phase = Magnesium
(x,y) = (-70, -41)
Euler = (357.5,86,101.2)
grainId = 929
	 index = 34279
	Id = 35721
	phase = Magnesium
	(x,y) = (-71, -40)
	Euler = (267.7,85.8,100.9)
	grainId = 929

Figure 27: `ebsd3` (see Table 8 for commands) *Selected point has **x=-70, y=-41** and Euler =(**357**,86,101)*	*Figure 28:* `ebsd4` *(see Table 8 for commands)* *Selected point has **x=-70, y=-41** and Euler =(267,86,101)*

Table 8: Various rotation commands and their results. Note each row starts with the same data, shown as Figure 25. By default, rotations are counterclockwise around the z-axis (which for this data is out of plane towards the viewer) as this command follows the right-hand rule.

`ebsd2=rotate(ebsd,90*degree,'keepXY')`	*Shown as* Figure 26*, spatial relationships are unchanged, but Euler angles are changed*
`ebsd3=rotate(ebsd,90*degree)`	*Shown as* Figure 27*, spatial relationships and Euler angles are both changed. This is the one to use if your map is slightly crooked.*
`ebsd4=rotate(ebsd,90*degree,'keepEuler')`	*Shown as* Figure 28*, spatial relationships are changed but and Euler angles are not.*
`ebsd5=flipud(ebsd)`	*(not shown) flips the map vertically, affecting both the spatial and Euler angles.* *Selected point has x=-40,* ***y=24*** *and Euler =****(273,94,281)***
`ebsd6=fliplr(ebsd)`	*(not shown) flips the map horizontally, affecting both the spatial and Euler angles.* *Selected point has* ***x=40****, y=70 and Euler =****(241,0,119)***

Out-of-plane rotations

Sometimes you need more complex rotations and are not just rotating the data in plane. Out of plane rotations are intended only for orientation rotations (e.g. pole figures), and not supported for spatial data (e.g. EBSD map plotting) but do seem to work anyway.

Out-of-plane rotations foreshorten the map perpendicular to the axis you rotate around. You may find this useful if you need to compensate for incorrect tilt correction during map acquisition. Due to the unsupported nature of this command, it may be a better approach to rotate only the orientations (using the keepXY tag demonstrated in Table 8 and scale the map to foreshorten it instead (see).

When would I need to rotate orientations out of plane?
When creating Figure 33 I wanted to compare pole figures from two regions, and needed to compensate for a known rotation and tilt in the friction stir welding process that passed through both regions as it sheared the material. Once I had done this, I could plot that data as two pole figures calculated perpendicular to the tool surface as it moved through the material and compare the pole figures in each location. In Figure 33, each pole figure at the right hand side of lines 1-3 and the left hand side of lines 4-5 was individually rotated based on the scan location.

Defining more complex rotations

In Table 8 we defined rotations as part of the `rotate` command, now let's use a variable to define the rotation, and then use the variable in the `rotate` command. Below are some examples of this method of defining a rotation. Note that in addition to the methods listed here you can also define rotations by Quaternions, Matrixes, Rodrigues Frank vectors, and other methods. See the online help functions for more detail on that.

Note that rotations are always defined relative to the sample, and not to the crystallographic axes. **If you're working with defining twins, or anything related to *crystallography* you should be working with orientations and not rotations.** See

Defining rotations using an axis-angle pair

The axis can be any vector3d. I find this an intuitive method of handling rotations and use it for most applications.

```
%define a rotation around y by 37 degrees
rot1 = rotation.byAxisAngle(yvector,37*degree)
```

Defining rotations using Euler angles

A rotation can be described by Euler angles. This makes sense, because an Euler angle defines how an EBSD point should be rotated to align with the default crystallographic axes. Two important points: the default Euler input is Bunge, so you don't have to specify that, but due to the way MTEX defines rotations, if you're copying something in the literature that rotates by Euler angles you need to swap the first and third angle input in the command since MTEX uses a different convention. Note that this does not affect Euler *Orientations*.

```
rot1 = rotation.byEuler(30*degree,50*degree,10*degree)
```

Defining rotations by four vectors

This type of definition is particularly useful when trying to line up your data to match a predefined axis. You have two vector3d in the start data which are rotated to match another set of two vectors. You can actually just use one vector before rotation and one after, and MTEX will calculate the minimum rotation to map the first vector onto the second.

Concatenating rotations

For more complex situations multiple rotations can be concatenated. Note that order matters here, and rotations are applied from the right to the left. So below, rot1 (defined in the section above) is applied to the data, followed by rot2.

```
%define a second rotation
rot2 = rotation.byAxisAngle(yvector,48*degree)
%calculate the combination of the two rotations
rot=rot2*rot1
```

Note that in general, rotating first in one direction and then second direction WILL NOT give the same result as concatenating the rotations! This is because after applying the first rotation, the relationship between your data (which just moved) and the axis of rotation you are using to define things (which is fixed) has changed. The concatenated rotation compensates for this by doing the math before moving anything, the sequential method does not, and use the original sample axes for everything.

Now we've defined the rotation, we apply it to the data, using the tags `'keepEuler'`, `'keepXY'` or neither tag as appropriate(see Table 8 for an explanation of these).

```
% rotate the ebsd Euler angles, and store this as a new variable
ebsd_rot = rotate(ebsd,rot,'keepXY')
```

This altered data can now be plotted (not shown).

Changing the displayed map without changing the data.

The following commands change **how the next** EBSD maps or pole figures will be plotted but have no effect on already plotted maps or your data. Your `ebsd` variable xy coordinates and Euler angles are both unchanged.

Changes made by using these commands reset every time you open MATLAB, so if you require these commands frequently I suggest changing the MTEX display defaults instead (see) which will be used with all plots until you update your MTEX installation.

To change a current figure without replotting it, go to the MTEX dropdown menu at the top of the figure window and change the selection for X axis direction or Z axis direction

- `plotx2north`
- `plotx2south`
- `plotx2east`
- `plotx2west`
- `plotzOutOfPlane`
- `plotzIntoPlane`

Scaling or translation (shifting) of data

To get the size of your EBSD map in µm (which is generally set based on information in the original imported file) use the following set of commands:

```
% calculate the x span and y span
xspan=max(ebsd.x)-min(ebsd.x)
yspan=max(ebsd.y)-min(ebsd.y)
```

However, if your data acquisition system has problems (see) you may find this value to be incorrect. You can easily correct point spacing for your `ebsd` variable without going back and altering the original .ctf or other input file.

```
%scale the x data down (by multiplying by a decimal), and shift it (by adding
a constant);
ebsd.x=ebsd.x*0.9108+0.0255
```

Note that if you're trying to flip a map, multiplying spatial values by -1 (e.g. `ebsd.x=-ebsd.x`) changes the spatial coordinates but not the axes. As a result, the new map will plot as a mirror image of the original version.

If you prefer to make changes to the spatial data by editing the .ctf manually, note that both the file header and the xy coordinates of each point must be updated, the latter of which can be done with an Excel macro.

Affine Transformation

An affine transformation performs a shear type displacement on a map. It rotates, translates, scales, and shears your data. In other words, it can turn a square into a rectangle or a parallelogram, but parallel sides remain parallel.

<u>When would I use an affine transformation?</u>
The distorted map of Figure 14 and Figure 15 demonstrate a case when an affine transformation would help you correct some data and render it usable for analysis. You'd need to measure the angles of the sides, but then altering the data is straightforward.

Since I've already covered and , which can be done with simpler methods, I'll just cover shearing of data.

For more information on the math of an affine transformation, try https://people.gnome.org/~mathieu/libart/libart-affine-transformation-matrices.html

An affine translation matrix for shearing looks like the following:

$$A = \begin{pmatrix} 1 & m1 & 0 \\ m2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

With the side of the parallelogram having a slope of 1/m. Therefore if you have a parallelogram where theta=15°, tan theta = 0.267, and the transformation matrix you'd want is probably shown at A2 in Table 9. Some other sample affine transformations are shown in this table as well, to demonstrate how the matrix affects the results. A less extreme real-world example was shown previously as Figure 14.

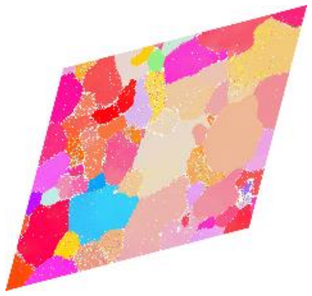
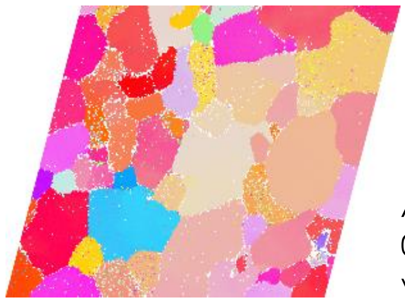
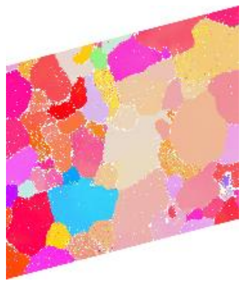
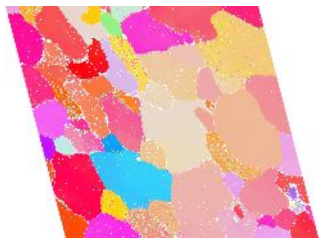
 A=[1,0.26,0;0.26,1,0;0,0,1] %value for both m1 and m2	 A2=[1,0.26,0; 0,1,0;0,0,1] % value for m1 but m2 set to zero
 A3=[1,0,0;0.26,1,0;0,0,1] %value for m2 but m1 set to zero	 A4=[1,-0.26,0;0,1,0;0,0,1] %negative value for m1 and m2 set to zero

Table 9: Four different affine transformations and their matrices

Once you've defined your affine transformation matrix apply it to your `ebsd` variable with the following code:

```
ebsdSheared = affinetrans(ebsd, A)
```

While the affine transformation updates the xy coordinates of the data, it doesn't change the Euler angles. Fortunately, this is unlikely to be required.

Reducing your data

Sometimes you need to resample your data to reduce file sizes. This is especially useful when testing time intensive analysis scripts. As an alternative to this, you can quickly crop your data (see)

```
%take every 8th point, not a random sample
ebsd_tiny=ebsd(1:8:length(ebsd))
```

Or alternatively

```
%take a random sample of 25% of the ebsd variable.
ebsdSub = ebsd(discretesample(length(ebsd),round(length(ebsd)*25/100)));
```

Data denoising and filling in holes

There are several filters that can be used to remove noise from your data (Mean, Median, Kuwahara, Spline, Halfquadratic, etc.) or fill in holes.

Removing noise means taking the existing orientation data and smoothing it so that there aren't abrupt changes within the same grain. In other words, the goal is to remove noise added to your data by the errors of the EBSD acquisition system. **Filling in holes** means replacing not-indexed points with indexed points by making educated guesses as to what they should be. These guesses are based on the surrounding points in one way or another.

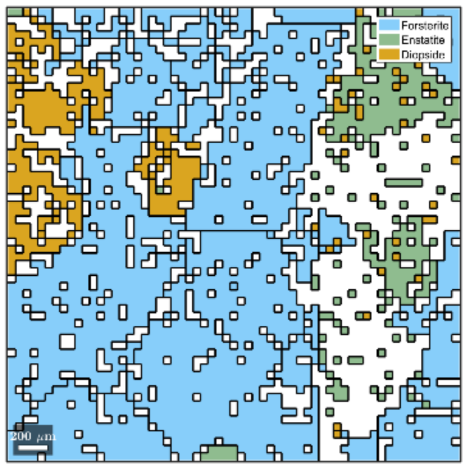
It's important to note that denoising and hole filling are two separate functions, although they can be done simultaneously. In all cases, keep in mind that you are altering your raw data. Be sure that you are altering it in a way that you want, and remember **the rule "garbage in, garbage out" applies strongly here**. When done properly, you get the results shown in Figure 29.

While there are several filters that can be used, we will only discuss the most straightforward: the mean filter. This filter fills points based on the average of that point's neighbours. However, if one, or several of the nearest neighbours are part of another grain, we don't want to average with them-this would move the grain boundary, create artifacts near the boundary or even merge the grains. Therefore, the best procedure involves calculating the grain boundaries before smoothing. For more details on , see that section later but for now just follow the code below. Note that the plots for Figure 29, 30, and 31 were created with the following commands :

```
figure; plot(ebsd); hold on; plot(grains.boundary,'linewidth',2)
```

or

```
figure; plot(ebsdS); hold on; plot(grains.boundary,'linewidth',2)
```



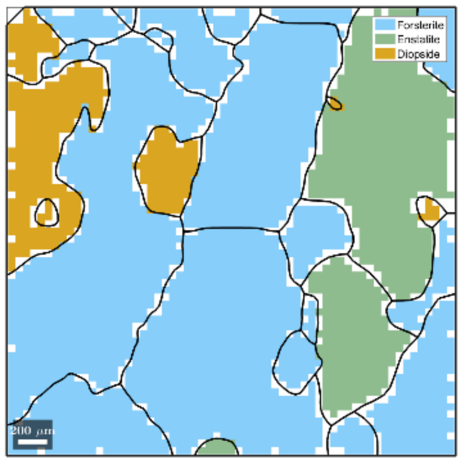
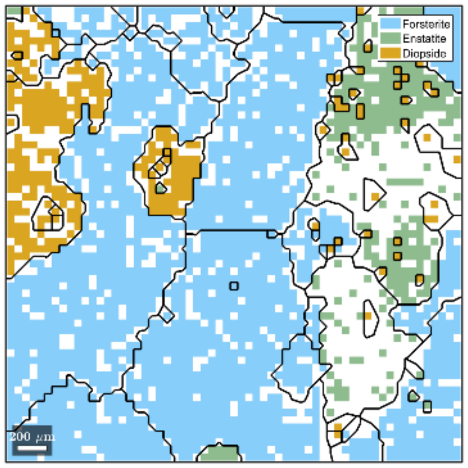


Figure 29: The effect of data denoising and hole filling. Left: before. Right: after filling. Note that the unindexed points directly on the grain boundaries are unaffected.

```
% import the MTEX test dataset named 'small' which is a subset of the
forsterite test dataset.
mtexdata small

%if you calculated the grains at this point, and then overlaid them on
the grain boundaries, you'd get the left hand side of Figure 29.  Note
that the white unindexed points have boundaries around them.
Essentially, they're treated like their own phase.

% reconstruct the grain structure.  When we do this, Let's consider
only the indexed data, in other words, we tell MTEX that the unindexed
data is 'empty' inside the grains rather than another phase.
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
```



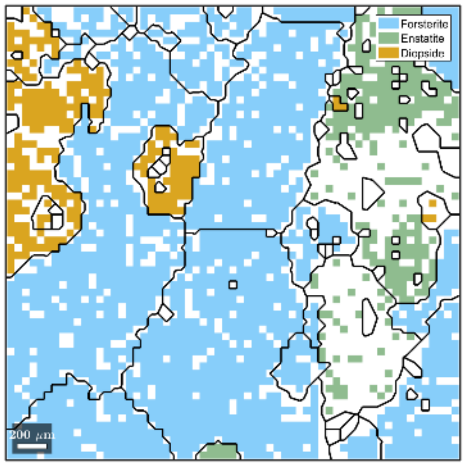


Figure 30: Left: The EBSD data sorted into grains, not including the unindexed points. Right, EBSD data within grains < 3 pixels was deleted (the grain boundaries are unchanged).

```
%at this point, if we plotted the ebsd data and the boundaries, we'd
get the result shown as Figure 30 left side.  Compare this to the left
side of Figure 29.  The difference is that the unindexed points are no
longer treated as a separate phase.

%We've still got some very small grains that are only one pixel big.
Let's remove all ebsd data belonging to grains less than three pixels
(see Figure 30 right side).  Note that you're actually deleting data
here (although the boundaries haven't updated yet).
ebsd(grains(grains.grainSize<3)) = [];

% redo grain segmentation.  Shown as Figure 31 left, the boundaries
around the grain < 3 pixels are now gone.
[grains,ebsd.grainId] = calcGrains(ebsd('indexed'),'angle',10*degree);
```

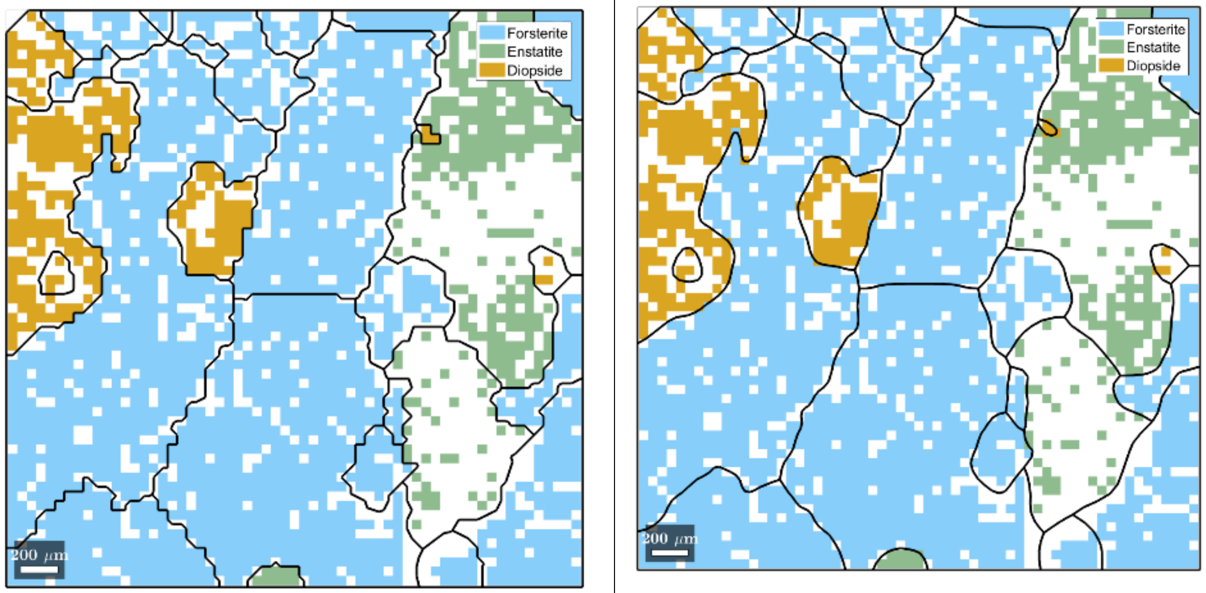
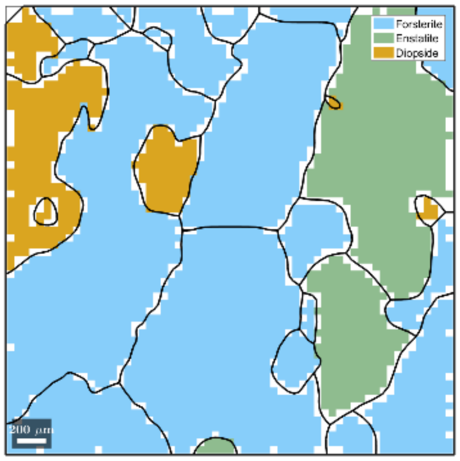


Figure 31: (left) Boundaries around small grains are removed and (right) smoothed

```
% smooth grain boundaries (see Figure 31, right)
grains = smooth(grains,5);

% define the filter to be used as meanFilter, and the number of
neighbours to consider
F = meanFilter;
F.numNeighbours = 3;

% smooth the data, using the filter specified.  Also, fill in the
holes, keeping the grain boundaries in mind. This was shown as the
right side of Figure 29, and again as Figure 32 (left side).
ebsdS = smooth(ebsd('indexed'),F,'fill',grains);
```



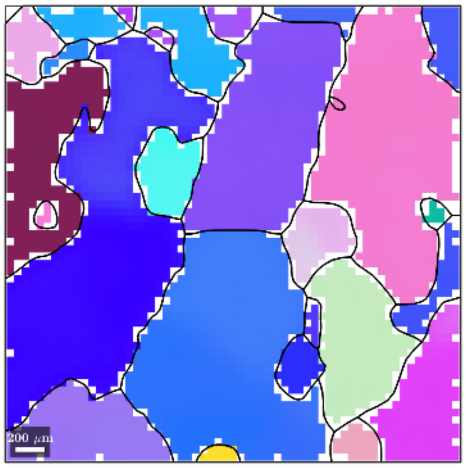


Figure 32: The smoothed data. (left) plotted by phase and previously shown as Figure 29 (right) Plotted by orientation; note the colour change within the blue grain at left from top to bottom.

You can also plot this by orientation rather than phase, shown as Figure 32 (right side). The most rigorous way to do this is to plot each phase one-by-one on the same map and then overlay the grain boundaries. Note that using the full phase name is not required, only enough to be a unique identifier.

```
figure;plot(ebsdS('f'),ebsdS('f').orientations)
hold on
plot(ebsdS('e'),ebsdS('e').orientations)
plot(ebsdS('d'),ebsdS('d').orientations)
plot(grains.boundary,'linewidth',2)
```

So, to sum up the steps in one place:

```
%calculate grains
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%remove EBSD data associated with small grains
ebsd(grains(grains.grainSize<3)) = [];
%recalculate the grains to remove boundaries around small grains.
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree)
%smooth the grain boundaries (optional)
grains = smooth(grains,5);
F = meanFilter;
F.numNeighbours = 3;
%smooth and fill the ebsd data
ebsdS = smooth(ebsd('indexed'),F,'fill',grains);
```

Sectioning spatially and re-connecting your data

There are all kinds of reasons to slice up your data spatially. Perhaps you want to compare the pole figures from two separate regions of your scan or want to crop out data from a scratch that created a series of mechanical twins that don't represent your sample.

One note of caution with slicing up your data: if you calculate grain boundaries, and then section the underlying `ebsd` data, the associated grain boundary data is **not** sectioned since it's in a separate variable. You therefore either need to calculate the grain boundaries for the segmented data (recommended method), or split the grain boundary data separately (see 0 for an example).

Reconnecting segmented data

This is very straightforward but assumes that the maps are all correctly positioned relative to each other (based on xy coordinates).

```
ebsd_a=[ebsd_1a ebsd_2a ebsd_3a ebsd_4a]
```

If you're joining adjacent maps named ebsd1 and ebsd2 with non-aligning xy coordinates things are a lot more fiddly. First, shift one map to align with the other using the instructions in . You may want to write a short script which will move the data in x or y and re-plot the map. This will save entering commands repeatedly by hand.

Next, crop out the overlapping area from one of the maps. A good reason to remove the overlap is so that if you create a pole figure or use the EBSD points for calculations you don't have a double contribution from the overlapping data.
The best way to delete the overlap is to get the x and y boundaries of both maps (e.g. `min(ebsd.x)`), calculate the overlap, and define the boundaries of that rectangle. You can then use these positions with the script in to delete that data. Finally, combine the maps into one `ebsd` variable for convenience. Obviously, this whole process would be a lot easier if your system recorded their relative position.

<u>Why would I want to reconnect segmented data?</u>
Aside from the obvious case of stitching two EBSD maps together, this can be useful for simplifying analysis. Imagine you have 100 small area scans (that aren't adjacent) and want to create one pole figure with all the data. You can create an `ebsd` variable that includes all the tiny maps, and then work with that. You could also combine your small datasets in various ways or rotate some of them relative to the others before recombining.

Manually selecting a rectangular subset

```
%load a test dataset
mtexdata twins
%plot the map to work from
```

```
figure;plot(ebsd('Magnesium'),ebsd('Magnesium').orientations)
%select two coordinates representing opposite corners of a rectangle
on the active EBSD map
[x,y]=ginput(2);
%get the length of rectangle sides using the x y values from ginput.
xspan=max(x)-min(x);
yspan=max(y)-min(y);
%use these corners to define a rectangle
region = [min(x),min(y),xspan,yspan];
%create a logical of points inside the rectangle
condition = inpolygon(ebsd,region);
%Use the logical to transfer the ebsd data within the rectangle to a
new variable.
ebsdSmall= ebsd(condition);
%plot the smaller data set.
figure;
plot(ebsdSmall('Magnesium'),ebsdSmall('Magnesium').orientations);
```

To view the square you selected, select the large map, then

```
hold on
rectangle('position',region,'edgecolor','r','linewidth',2);
```

If you want grains data for your reduced dataset, the fastest way is to calculate it fresh. But if you need to segment it the syntax is provided below.

```
%if you haven’t done it already calculate the grain variable for the full
data set
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%Create a logical of grains data within the segmented data boundary.
[in, ~] =
inpolygon(grains.boundary.midPoint(:,1),grains.boundary.midPoint(:,2),[max(x)
min(x)], [min(y) max(y)]);
%segment the grains data to match the spatial segmentation
bound_segment=grains.boundary(in);
%plot the result
figure;plot(ebsdSmall('Magnesium'),ebsdSmall('Magnesium').orientations);
hold on
plot(bound_segment,'linewidth',2)
```

Manually selecting a polygonal subset

While you could use four points to define the corners of a rectangle or square, they won’t necessarily be parallel. So this code is best for free form shapes. Note that points are drawn on the map as you select them.

```
%load some data
mtexdata twins
%plot a map if it doesn't exist
figure;plot(ebsd('Magnesium'),ebsd('Magnesium').orientations);
```

```
%select a series of x,y points, close the shape with a right click.
poly = selectPolygon
%create a logical identifying points as inside or outside the polygon
ind = inpolygon(ebsd,poly);
% reduce the data to that within the polygon
ebsd_poly = ebsd(ind)
% show the selected polygon region
figure;plot(ebsd_poly('Magnesium'),
ebsd_poly('Magnesium').orientations);
```

Sectioning to vertical or horizontal slices or a grid

This code macro is available via the script repository (link below or google jhiscocks gist and look for MapSeg).
https://gist.github.com/jhiscocks/2f4b4bd51bf44d68bdd23ebfd5f7d10e

When is it useful to slice your data into small squares or strips?
I used this script frequently for my article *Influence of Magnesium AZ80 Friction Stir Weld texture on Tensile Strain Localisation* to convert thin and long scans (10 µm by 1 mm) to series of pole figures at various points relative to an interface as shown in Figure 33. This script was also the basis for the grid of vectors shown in Figure 34.

To create the grid of vectors in Figure 34 after data was 'binned' into a grid of subsets. A script was used to perform the following operations on all subsets:

- convert the subset to an Orientation Distribution Function (ODF).
- Extract the most intense orientation for each subset using the `mode` function
- convert the orientation to a vector3d representing the c-axis
- resolve the vector3d along each of the x, y, and z axes, and store the resolved vector components in an array with the x and y position of the subset
- use the 3d quiver plot function to the array as a grid of arrows.

This kind of work requires scripting (especially because you need to play around with subset size), so would not be feasible in analysis software that does not allow for batch processing and scripted plotting of results.

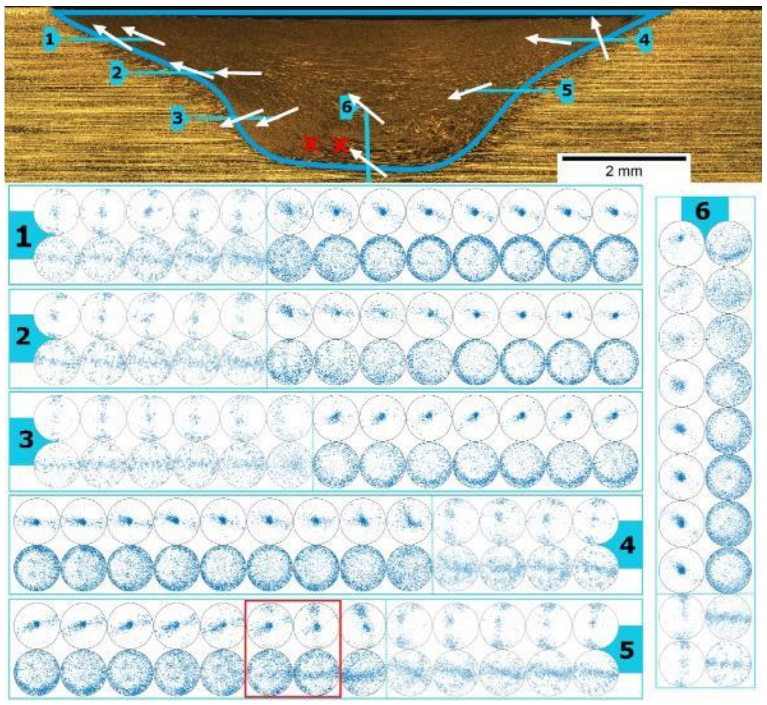


Figure 33: Line scans across the boundaries of a friction stir weld, segmented into pole figures.

0Plotting misorientations along a line

It's very common to want to draw a line on a previously plotted EBSD map and see how the orientations change along it and across grain boundaries. Note that because you can't plot a set of Euler angles along a line (how would that even work?) these plots actually compare one point (usually the start of the line) to all the others along it. This comparison is typically in the form of the misorientation angle between the reference orientation and the rest of the data.

The misorientation angle is the minimum angular rotation to bring two crystallographic unit cells into alignment. Therefore, in order for the calculation to be valid, all points along the line must have the same unit cell/phase, and so the commands below will generate an error message if the line crosses more than one phase. In that case, you need to split the data and line to include only a single phase.

```
%load a magnesium test data set
mtexdata twins
%plot a map if it doesn't exist
figure;plot(ebsd('Magnesium'),ebsd('Magnesium').orientations);
%select endpoints of a line on the map and store the x and y coordinates
[x,y]=ginput(2);
```

```
% create a variable holding the xy spatial coordinates of each end of the
desired line segment
lineSec =[x(1,:)  y(1,:); x(2,:)  y(2,:)];
%overlay the line you selected on the ebsd map as a visual reference
hold on;line(x,y,'linewidth',2); hold off;
%Send the line coordinates to the function spatialProfile, which will return
the EBSD data of points on the line and the distance from the first point
[ebsdLineOri,dist] = spatialProfile(ebsd('Magnesium'),lineSec);

%calculate the misorientation angle between the first point of the line (your
reference) and all the points on the line
ang= angle(ebsdLineOri(1).orientations,ebsdLineOri.orientations)./degree;
% plot this misorientation angle vs the distance along the line
figure;plot(dist,ang);
xlabel('Distance along line'); ylabel('misorientation angle in degrees')
```

Optional: on the same plot, compare each point on the line to the previous point

```
% to plot the misorientation gradient, we calculate the misorientation
between each adjacent point on the line instead.  Note that we have to
truncate the variable dist by one in order to make the variable lengths
match.

ebsdLineOriDelta= angle(ebsdLineOri(1:end-
1).orientations,ebsdLineOri(2:end).orientations)./degree;
hold on;
plot(dist(2:end),ebsdLineOriDelta);
hold off
%label your plot axes
xlabel('Distance along line'); ylabel('misorientation angle in degrees')
%add a legend
legend('misorientation to first point','orientation gradient')
```

Specialty map plots: really using MTEX

As we discussed earlier in the section the first part of the plot command provides xy coordinates for the spatial location of points of your map, and the second part of your command provides values used to colour each point. Previously we used scalar data stored in the `ebsd` variable to provide this colour info, but that is not required. Colouring with orientation data is covered in section .

In terms of input, the colour information basically requires either a vector array, or set of orientations the same length as the `ebsd` data used for spatial data. So if you're working with the forsterite dataset and plotting the whole `ebsd` variable you can check how long your colour-providing vector array needs to be with the code below (the answer is 187467).

```
mtexdata forsterite
size(ebsd('Indexed'))
```

Therefore to make the plot command `plot(ebsd('Indexed'),colourInfo)` work, the variable `colourInfo` needs to be an array 1x187467 or 187467x1 (either a row or column vector array are both fine). If it's larger, for example if `colourInfo` is a 3x187467 array you need to specify a portion of the variable to use (with syntax like `colourInfo(:,2))` so that only a portion of the variable is used with the plot command.
A common approach is to do a series of complex calculations on your EBSD data, resulting in a variable that can be used for colouring it, and input that into the plot command to see the visual representation of these calculations.

```
%load the forsterite test data
mtexdata forsterite
%create a vector array containing numbers from one to 187467
colourInfo=1:187467;
%Plot all indexed data, with an ascending colour value (from the
ascending values in the variable named colourInfo).
figure;plot(ebsd('Indexed'),colourInfo)
```

Note that if you were just plotting one phase of this dataset then the length of `colourInfo` would need to be shorter to match it (e.g. if you were plotting Diopside from the forsterite datset, `colourInfo` must be exactly 9064 entries long). Similarly, if you're getting the xy data from the grains, the colour variable should match the length of the XY grains data.

```
%calculate grains
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%plot the grains, coloured by the grain ID number.  Note that both
parts of the variable are 3156 entries long (for the forsterite
dataset if the threshold angle was set at 10 degrees using the above
command.
figure;plot(grains,grains.id)
```

A common application for this approach is calculating the Schmid factor of various slip systems and plotting the results.

Mapping by Schmid Factor

Colouring maps by calculation outputs is commonly used for Schmid factor maps. A script to calculate Schmid factors and incorporate critical resolved shear stresses for magnesium is online at the script repository (link below or google jhiscocks gist and look for SchmidR2). To use that function for another material you'll have to know the slip planes and directions for all systems you're interested in.
https://gist.github.com/jhiscocks/6912714e30721908f213887d50341158

One reason that this type of script is hard to customize is that it is a very specific calculation. In fact, even different phases of the same base element will need their own customized script.

Colouring an EBSD map by IPF in various directions

In MATLAB, the default colourmap for numeric (scalar) data, is named `parula`. Your options for numeric plots like this are covered in . For orientation maps, (which are coloured by Euler or other multi-parameter angles) you need an actual 2D colourmap, which is handled differently and discussed in this section. The code to view that map for a given phase is as follows (assuming you have some EBSD data loaded):

```
%create the ipf sector/shape and color it
oM = ipfHSVKey(ebsd('Magnesium'));
%view the orientation map used
figure;plot(oM)
```

The figure plotted with these commands (previously shown as Figure 21) describes the conversion between crystallographic orientation and colour and is called an orientation map. This map depends only on the phase (although the colour distribution can be customized) and so the previous commands are only required once per MATLAB session. Now let's apply the default orientation map to an EBSD plot, colouring all the points by their orientation.

To do this, we need to also define a vector3d. What the software does for each orientation point is look at what plane of the point's unit cell is oriented closest to the vector3d, and then colour it based on the orientation map. So for magnesium any red parts of the map have the [0001] axis oriented in the vector3d. In other words, red means that the [0001] orientation aligns with the selected vector3d specimen direction. If no vector is defined (i.e. if you skip the first line of the code below) the default vector3d is in the z direction.

```
%define y as the vector3d direction.
oM.inversePoleFigureDirection = yvector;
%convert the ebsd map orientations to a color based on the IPF
color = oM.orientation2color(ebsd('Magnesium').orientations);
%plot the figure using the new colour map
figure; plot(ebsd('Magnesium'),color);
```

Altering the vector3d chosen is easy, and you don't need to repeat most of the commands

```
oM.inversePoleFigureDirection = xvector;
color = oM.orientation2color(ebsd('Magnesium').orientations);
plot(ebsd('Magnesium'),color);
```

Any vector3d can be selected, one example of which is shown below. Some other examples of different ways to define vector3D are detailed in .

```
%define a vector3D at 45 degrees between the positive x and negative y
specimen directions
V45 = vector3d(1,-1,0);
%use this as the IPF direction
oM.inversePoleFigureDirection = V45;
%calculate the colours for each of the orientations.
color = oM.orientation2color(ebsd('Magnesium').orientations);
%plot the map using this colouration
plot(ebsd('Magnesium'),color);
```

In general, the default orientation map colouring is fine for most data. However, we can also alter it, controlling what colours are used for what orientations. For example, if we have an EBSD scan that was limited to one large grain, the default colourmap would be fairly useless; everything would be very similar colours due to the low angular change between points, so you'd want a small change in orientation to show a larger change in colour. For details on how to do this, look up *Sharp Color Keys* in the online documentation, or follow the example in .

Colouring by angle of c axis to a selected reference

As discussed in , maps can be coloured by calculation outputs. Figure 34 (left side) shows an example where all orientations were converted to vector3d of the c axis [0001] and all compared against a reference (the 'X' mark). This type of map is particularly useful for non-cubic systems, because it makes the change in orientation across the sample very easy to visualize. In Figure 34, it makes the periodic change of texture along a friction stir weld easy to visualize (from my article *Formation mechanisms of periodic longitudinal microstructure and texture patterns in friction stir welded magnesium AZ80)*.

The creation of the grid of vectors shown at the right side of Figure 34 is described in the section , and is an excellent example of the kind of thing that's (relatively) straightforward in MTEX but impossible in other software that doesn't support scripting and arrays. I developed this particular plot method myself, and I'm quite proud of it.

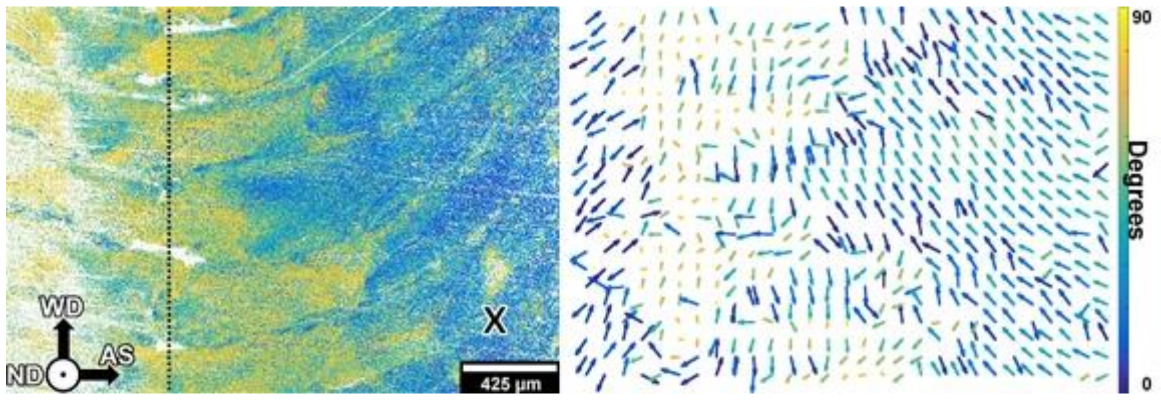


Figure 34: EBSD data coloured by angle between the local c-axis and that of a reference point (marked with an 'x'). For explanation of the vector plot, see

Here are a couple of ways to create a plot like the left side of Figure 34:

```
%load a magnesium sample data set
mtexdata twins
%plot the map
plot(ebsd('Magnesium'),ebsd('Magnesium').orientations)
%save the crystal symmetry info to its own variable for convenience
cs= ebsd('Magnesium').CS
```

OPTION 1: Using a point from your map for comparison. This is what was done for Figure 34, and compares the angles between the c-axis of two EBSD points. Note that this is not a misorientation calculation since you're comparing two axes rather than two unit cells. If you want to do misorientation angles, see the note at the end of this section.

```
%Collect the x-y coordinates of an ebsd point by clicking on the map
[x,y]=ginput(1);
%convert the selected orientation to a vector3d representing the c axis
direction
ori1=ebsd(x,y).orientations*Miller(0,0,0,1,cs,'HKIL')
```

OPTION 2: Using a vector3d for comparison.

```
%compare to one of the axis direction vectors
ori1= yvector
```

Now you have a reference vector for comparison, both option 1 and 2 continue the same way; you need to compare all your `ebsd` points to this reference.

```
%use the orientations of all ebsd points to calculate a vector array of
vector3D representing the c-axis directions.  This will be the same length as
the ebsd variable.
ori2= ebsd('Magnesium').orientations*Miller(0,0,0,1,cs,'HKIL');
```

```
%calculate the angle between ori1 (a single vector3D) and each vector3D
stored in the variable ori2.  The antipodal command means that the max
allowable value is 90 degrees rather than 180 (i.e. both ends of the c axis
are equivalent).
ang=angle(ori1, ori2,'antipodal')/degree;
%plot the map.
figure; plot(ebsd('Magnesium'),ang);
```

If you picked option 1 above, the grain you selected ori1 in will be coloured to show low values (dark blue by default), since all the points in ori2 that make up this grain are very close to the reference angle (i.e. ori1)

Note that to calculate the **misorientations** between the reference point and all the `ebsd` points, we need to use two orientations as inputs (i.e. don't multiply either the reference point or the rest of the EBSD data by the miller indexes). The maximum possible value of `ang` increases to ~93 degrees for magnesium because this is a misorientation now. The maps will look almost identical but typing `max(ang)` and comparing the result should show a small difference.
This type of plot can also be done for grains (see)

Defining grains and using them

To use MTEX well, in addition to the `ebsd` variable you also need to understand the grains variable, which is usually created manually by the user at some point after EBSD data has been imported. **Note that when you generate the grains variable, certain properties are usually also added to the `ebsd` variable.** Information added to the `ebsd` variable includes the ID of the grain that each EBSD point belongs to, and even deleting the `grains` variable will not remove those properties appended to the EBSD variable. If you look at the command below, used to create grains you can see this is an MTEX function named `calcGrains`. The `ebsd` variable and a numerical threshold are passed to it in the right-hand side brackets, and the function returns a new variable named `grains` and appends two properties to the `ebsd` variable at left.

```
%load the magnesium test data
mtexdata twins
%define some grains, using a threshold of 10° for grain boundaries
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
```

Just like with the `ebsd` variable, each `grains` entry has an ID, assigned sequentially when the variable was created. Remember that when using syntax like `grains(125)` you are calling the 125th listing in the `grains` variable, and NOT using the ID number. Using the ID number is the more robust way to do things, and the two methods are only equivalent until one grain is deleted. At that point, all bets are off. To reference by ID, you need something like the following:

```
temp=grains(grains.id==125)
```

Recently, MTEX has added the capability to also calculate sub-grains. The syntax for that is fairly straightforward (see below) and more details about that may be seen in the online documentation.

```
[grains,ebsd.grainId] =
calcGrains(ebsd('indexed'),'threshold',[1*degree, 10*degree]);
```

Remember, that everything related to point orientations is still in the `ebsd` variable, and hasn't moved. The `grains` variable only stores one orientation per grain: the mean value. So for many types of plots you'll use both the `ebsd` and `grains` variables. For example, if you've calculated grains, and you now want to take the largest grain, and plot only the orientations within that grain you would use the following code:

Option 1 (steps broken out to help you customize):

```
%Find the area of the largest grain
temp1=max(grains.area)
%create a logical of grains that have their area equal to the max area
temp2=(grains.area==temp1)
%Use the logical generated to filter the grains variable.
temp3=grains(temp2)
```

Option 2:

```
%perform all the steps above in one command
temp3=grains(grains.area==(max(grains.area)));
```

Option 3:
There's actually a helpful shortcut for the above command, which returns more information back from the max function.

```
%get the maximum area of the grains, and the grain id for where it exists
[val,temp3] =max(grains.area);

%now you need to get the EBSD data.  You use the grain Id (from the
grain stored in temp3) to create a logical of all ebsd points having
that grainId, then you use that as a filter on the ebsd variable.
temp4=ebsd(ebsd.grainId==temp3.id)
%plot the grain as a figure, coloured by band contrast
figure; plot(temp4,temp4.bc);
```

Each grain has a number of sub-properties, including all information associated with the boundaries surrounding it and associated subgrains. These properties can be very useful for creating plots and performing calculations. Table 10 lists some of these, and a more inclusive list can be seen by typing `grains.` or `grains.boundary.` into the command line and hitting the tab key.

Table 10: Selected properties of the `grains` variable

`grains.area`	The area in scan units (usually µm)
`grains.aspectRatio`	the ratio between the longest and shortest axis of a grain
`grains.boundarySize`	Number of boundary segments+1
`grains.boundary`	The list of boundary segments. On its own, sums up segments by phase on each side. Includes much more information, see Table 11.
`grains.centroid`	Grain centroid x and y coordinates. Centroid may be outside grain perimeter (e.g. doughnut or 'c' shaped grains)
`grains.diameter`	the longest distance between any two vertices of the boundary in scan units
`grains.GOS`	Grain Orientation Spread; average misorientation to the meanOrientation
`grains.grainSize`	Number of pixels per grain
`grains.hasHole`	returns logical 1 if grain has hole
`grains.id`	the ID number of the grain
`grains.isBoundary`	Returns logical 1 if grain intersects scan boundary
`grains.meanOrientation`	The mean orientation of each grain
`grains.mineral`	grain phase as text
`grains.neighbors`	Two-column list, where each row is ids of two adjacent grains
`grains.numNeighbors`	Number of neighbouring grains
`grains.perimeter`	Perimeter in scan units
`grains.phase`	grain material phase as an integer
`grains.smooth`	A smoothed version of the grains
`grains.triplePoints`	Triple point locations

The grain boundary properties can be accessed either at the top level by `grains.boundary` (which provides statistics on all boundaries) or more helpfully at the individual grain level. Below is one example, but you can Try this with the other options listed in Table 11.

```
%get the ID of the grain with the highest orientation spread within it.
[~,idNUM] =max(grains.GOS)
%move that grain to a separate variable
temp=grains(grains.id==idNUM)
```

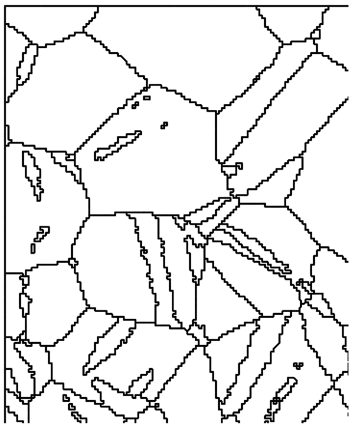
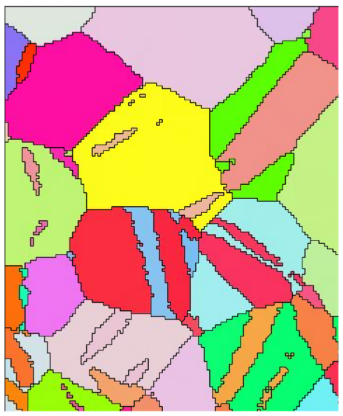
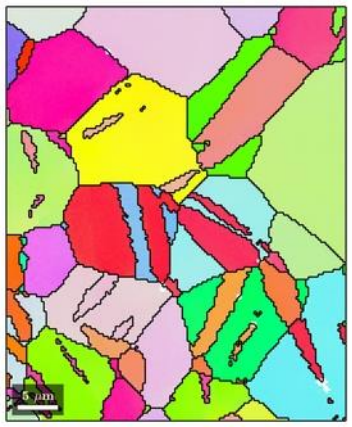
```
%get information about what grains share a boundary with the selected one.
Note that this outputs a list of pairs, and the value stored in the variable
idNUM will be included on each row.
temp.boundary.grainId
```

Table 11: Selected properties of the grain boundaries

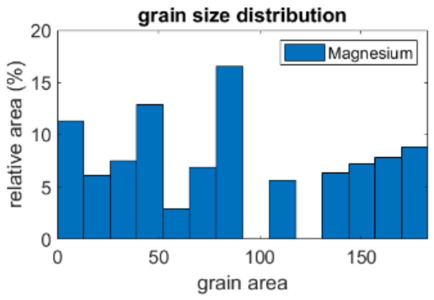
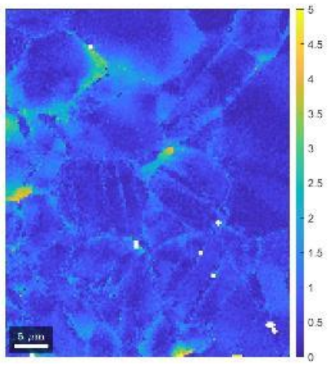
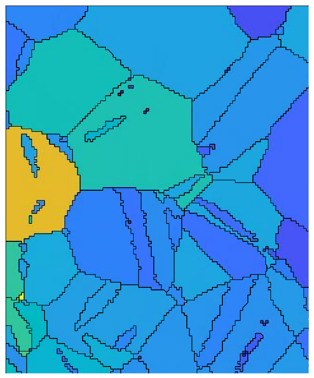
`grains.boundary.direction`	a vector3d representing the segment's direction
`grains.boundary.ebsdId`	the id numbers of the data points on either side. Currently this works only if you create grains from all the data, while it is incorrect if you only use the indexed data.
`grains.boundary.grainId`	The IDs of all grains on either side of the boundary
`grains.boundary.length`	Total count of all segments
`grains.boundary.midPoint`	The x y coordinates of the midpoint of all boundary segments
`grains.boundary.mineral`	the phase on either side of the segment as text BUT can only be used if one phase (e.g. if a zero sol'n on one side of the boundary, get error)
`grains.boundary.misorientation`	The misorientation angle across the boundary, can be converted to an axis and angle BUT only valid if same phase on each side
`grains.boundary.x or y`	Coordinates of the end of each boundary segment

Some grain-based plots

There are a lot of plots that can be made using the grain data, and the same syntax is used as for EBSD maps (see) Some common plots are shown in Figure 35. Some of the plots shown there draw from the `grains.boundary` data, some from the `grains` data directly, and some from the data that is appended to the `ebsd` variable when grains are calculated. Note that two of the plots in Figure 35 show orientation, one using the grain average (one value per grain) and the other using EBSD data (one value per `ebsd` point). Also shown in Figure 35 is a histogram, which was created with a special abbreviated command. The full command syntax is discussed in , and will allow you to customize the plot.

`% plot the boundary of all grains`
`plot(grains.boundary, 'linewidth',1.5)`
	`%if you only have one phase`
	`plot(grains, grains.meanOrientation)`
		`% for multiple phase plots`
		`plot(ebsd('magnesium'), ebsd('magnesium').orientations)`
		`hold on`
		`plot(grains.boundary, 'linewidth',1.5)`

`hist(grains)`	`%plot the misorientation of each point relative to the average value`
	`plot(ebsd('Magnesium'),ebsd('Magnesium'). mis2mean.angle./degree);`
	`caxis([0 5])`
		`%plot the span of misorientations within a grain in degrees`
		`plot(grains,grains.GOS./degree)`

Figure 35: A selection of grain plots based on the twins dataset

There are plenty of other ways to make grain-based plots, some of which are discussed here.

Colourizing using the basic properties of a grain

Most of the options in Table 11 should work as input on the right side of the following command. For plots like this that only call on scalar properties, the number of phases or specific phase referencing can be omitted.

```
% colorize grains according to area
plot(grains,grains.area)
```

Colourizing using the grain boundary properties

For plots like this, since orientation is involved, the phase needs to be explicitly stated.

```
% colorize grain boundary according to misorientation angle-a good way to
spot twins
%first, get all the boundaries that are Mg on both sides (because if they
aren't you cant calculate the misorientation across them
gB = grains.boundary('Magnesium','Magnesium')
%next use the misorientation angle to colour the plot.
plot(gB,gB.misorientation.angle)
```

IPF color for a grain mapped sample

This is exactly the same as , but done on the grain average orientation rather than point-by-point. As a result, this runs much faster but gives similar results since you can't really see changes of a degree or two anyway.

Option 1: working with single phase data

```
%create the ipf sector/shape and color it
oM = ipfHSVKey(grains);
%define the direction of the ipf
oM.inversePoleFigureDirection = xvector;
%convert the ebsd map orientations to a color based on the IPF
color = oM.orientation2color(grains.meanOrientation);
plot(grains,color)
```

Option 2: working with multiple phase datasets. I used the forsterite dataset and plotted the forsterite phase. To add the other phases, repeat the below commands, adding the `hold on` command before adding to the figure.

```
%create the ipf sector/shape and color it
oM = ipfHSVKey(grains('F'));
%define the direction of the ipf
oM.inversePoleFigureDirection = xvector;
%convert the ebsd map orientations to a color based on the IPF
color = oM.orientation2color(grains('F').meanOrientation);
plot(grains('F'),color)
```

As an exercise create two plots, one using the code above (drawing from the `grains` variable), and one other based on the orientations in the `ebsd` variable and compare.

Isolating a grain, and other grain-based subsets

Once grain boundaries are calculated, you can isolate grains by Id. Alternatively, you can extract multiple grains that meet a set of conditions.

```
%get the highest id number in the grains variable
max_id=max(grains.id)
%get the data of the grain with that id number
selGrain=grains(grains.id==max_id);
%plot that grain, and outline it.
figure;plot(selGrain,'linecolor','red','linewidth',1.5)
```

You can also set conditions to filter your grains

```
%get a logical of grains with a perimeter larger than 60 and a grain size
equal or bigger than 900
condition = grains.perimeter>60 & grains.grainSize >= 900;
%copy the grain information of these grains to a new variable
selected_grains = grains(condition)
%plot these grains
figure;plot(selected_grains,selected_grains.area)
```

Or manually select a grain from a map

```
%click on an existing map and get a point
[x,y]=ginput(1);
%get the grain containing these xy coordinates
selGrain=grains(x,y)
```

Find/select grains by orientation

Unlike the previous section, these commands need to be addressed to a specific phase.

```
%define an orientation
o2 = orientation('euler',0*degree, 0*degree, 30*degree,'ZXZ',ebsd('f').CS);
%select grains within twenty degrees of that orientation
grains_selected = grains.findByOrientation(o2,20*degree)
```

Note that instead of manually defining an orientation, you can use one from an existing grain.

```
%click on an existing map and get a point
[x,y]=ginput(1);
%get the grain
selGrain=grains(x,y)
%get all grains that are within 20 degrees of the selected grain’s
orientation
grains_selected =
grains.findByOrientation(selGrain.meanOrientation,20*degree)
```

Calculating grain size

This example walks you through the steps of calculating grain size and making some corresponding plots. Let’s start with the twins dataset. If you want to merge twins (and not count them as separate grains, see .

```
%load some data
mtexdata twins
```

```
%calculate the grains from (only) the indexed data
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);

%to remove all grains smaller than a certain size from the calculation,
create a logical the same size as the grain list.
temp=grains.grainSize> 2;
%now select the grains marked as 'true' in the logical we just created
selected_grains=grains(temp);

%check that this has worked correctly.
min(selected_grains.grainSize)
%you can calculate the mean area in um
mean(selected_grains.area)
%find out how many grains were used to calculate the mean
length(selected_grains.area)
%the map of grains used for this calculation can be seen with
figure;plot(selected_grains, selected_grains.area)

%you can now plot a histogram of the data in scan units
figure; histogram(selected_grains.area)
xlabel('grain area in scan units'), ylabel('Number of grains')

%OPTION: remove all grains that intersect the scan boundary (edge grains).
selected_INTgrains=selected_grains(selected_grains.isBoundary==0)
%the map of grains used for calculation can be seen with
figure;plot(selected_INTgrains, selected_INTgrains.area)
%you can calculate the mean area in um
mean(selected_INTgrains.area)
%find out how many grains were used to calculate the mean
length(selected_INTgrains.area)
```

Examining misorientation spread within a single grain

Let's say you want to take a closer look at one grain of interest.

```
%load some data
mtexdata forsterite
%calculate grains
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);
%Pick the grain with the largest orientation spread
[val,id] = max(grains.GOS)
grain_selected = grains(grains.id==id);

%Alternately
grain_selected=grains(grains.GOS==max(grains.GOS))

%and then plot the ebsd data for that grain only
plot(grain_selected.boundary,'linewidth',2)
hold on
plot(ebsd(grain_selected),ebsd(grain_selected).mis2mean.angle./degree)
hold off
mtexColorbar
```

```
%or plot it colorised by orientation, with the boundary outlined
figure;plot(grain_selected.boundary,'linewidth',2)
hold on
plot(ebsd(grain_selected), ebsd(grain_selected).orientations)
hold off
```

Customizing the orientation map to a specific grain

The orientation map is designed to be able to represent any possible orientation with a colour. Therefore, when you're looking at a single grain, the narrow spread of orientations shows up as mostly the same colour. Sometimes we want to increase the colour contrast for a single grain, even though this new colourmap won't be good for general usage or a different grain.

```
%Create an orientation map specific to the ebsd points of the grain we
selected.  This allows us to narrow the orientation map to better cover the
grain
ipfKey = ipfHSVKey(ebsd(grain_selected).CS.properGroup);
%take a look at the colour map which we're about to customize
figure;plot(ipfKey)

%center the coloring of the IPF on the mean orientation of the grain (you'll
get a warning here)
ipfKey.inversePoleFigureDirection =
mean(ebsd(grain_selected).orientations,'robust') * ipfKey.whiteCenter
figure;plot(ipfKey)

%shrink the range to be that of the value of the GOS (already in radians)
%since this is a 'spread' we multiply by 2, plus a bit to keep away from the
%edges of the colourmap
ipfKey.maxAngle = 2.5*max(grains.GOS);
figure;plot(ipfKey)

% plot the selected grain, with the revised orientation map with an outlined
boundary
figure;plot(grain_selected.boundary,'linewidth',2)
hold on
plot(ebsd(grain_selected),ipfKey.orientation2color(ebsd(grain_selected).orien
tations))
hold off
```

Defining twin boundaries

Many materials create twins due to mechanical or thermal stress. In twins, the crystal structure is mirrored about a certain plane, unlike grain boundaries where the misorientation axis and angle can be any possible value. Each metal will have a specific orientation relationship for each type of twinning that occurs in that metal. For example, in Magnesium you can expect to see many extension twins with a rotation of 86.29° about the {1,1,-2,0} axis, and (in rare cases) compressive twins with a rotation of 56° about the same axis. A different metal will have different twin relationships, and some metals do not twin at all. While this example defines twins by an axis-angle pair, a different definition is used in the section 0

```
%load the twins dataset
mtexdata twins
%calculate the grains
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);
%Copy the GB information to a new variable
gB = grains.boundary

%To not include the border of the EBSD map in calculations, create a new
variable for only the mg-mg segments
gB_MgMg = gB('Magnesium','Magnesium')
%which can then be plotted, colorised by misorientation if you want
figure;plot(gB_MgMg,gB_MgMg.misorientation.angle./degree,'linewidth',2)
mtexColorbar

cs=ebsd ('Magnesium').CS
%define an extension twin (86.29 degrees reorientation) by axis and rotation
%for magnesium extension twin
h= Miller(1,1,-2,0, cs)
twinning = orientation('axis',h,'angle',86.3*degree,cs,cs)
%for compression twin you would use the below command instead
%twinning = orientation ('axis',h,'angle',56*degree,cs,cs);

%Next we create a logical of the boundary segments within 5 degrees of the
twin relationship we just defined.
isTwinning = angle(gB_MgMg.misorientation,twinning) < 5*degree;
%locate the segments that meet this condition
twinBoundary = gB_MgMg(isTwinning)

% plot the grains, and overlay the twinning boundaries
figure;plot(grains,grains.meanOrientation)
hold on
plot(twinBoundary,'linecolor','blue','linewidth',2,'displayName','twin
boundary')
hold off

%to calculate percentages of twin boundary out of all boundary segments
100*length(twinBoundary)/length(gB_MgMg)
```

Merge twins along twin boundaries

Grains that have a common twin boundary are assumed to inherit from one common grain (the 'parent'). We often want to merge these twins with the other parts of the grain across these special boundaries. For example, when we make grain size measurements we want MTEX to consider both the twinned and untwinned parts of the grains.

To merge grains which have a common twin boundary together, we use the merge command. The `mergedGrains` variable has all the same properties as the `grains` variable presented earlier.

```
%continuing from the previous code example, let’s merge the already defined
twins with the parent grains.
[mergedGrains,parentId] = merge(grains,twinBoundary);

% plot the merged grains
figure;plot(ebsd('indexed'),ebsd('indexed').orientations)
hold on
plot(mergedGrains.boundary,'linecolor','k','linewidth',2.5,'linestyle','-
',...
  'displayName','merged grains')
hold off
```

After doing the following, note that the twins no longer have a boundary around them. Also, compare the length of the `grains` variable with the length of the `mergedGrains` variable, which has fewer entries.

However, although MTEX can find all the twin boundaries based on an angle-axis definition, the software has no way to tell which is the original grain (i.e. the parent), and which is the twin which was formed later. An experienced metallographer can usually tell, because these twin boundaries are often more straight than normal grain boundaries. A script is available to manually flag the parent based on visual observation, or automatically assume that the larger section is the parent; (link below or google jhiscocks gist and look for Parent_Twin_InteractR18 or Parent_Twin_Area). These scripts also generate tables of information about the parent grains and twins.

https://gist.github.com/jhiscocks/274ae179303280e00749440698f7f0ac
https://gist.github.com/jhiscocks/4acc046799d90dfe54e01936527a51e5

Calculate the twinned area

We can also calculate the area percent of grains that have a twin within them. For more accurate twinning estimates, see the script available online.

```
%Using all twinning boundary segments, get the grain ids on either side
(reduced to unique values).
twinId = unique(gB_MgMg(isTwinning).grainId);
% compute the area fraction
sum(area(grains(twinId))) / sum(area(grains)) * 100
%visualize the untwinned grains (in colour)
figure;plot(grains,grains.meanOrientation)
hold on
plot(grains(twinId))
hold off
```

Maps Colorised by calculations involving grains

Coloring grain map by angular deviation of c axis from a selected grain

A version of this using EBSD data is shown earlier as .

```
%get the ID of the largest grain (this function returns the value and grain
ID)
[~,bigGrainID]=max(grains.grainSize)
%get the meanOrientation
ori1=grains(grains.id==bigGrainID).meanOrientation;
%calculate a vector array of all the grain meanOrientations
ori2=grains.meanOrientation
%determine the misorientation between the grains and the reference in degrees
mori=angle(ori1, ori2)./degree;
%plot the map on a greyscale, note that the largest grain (our reference) is
black.
figure; plot(grains,mori);
colormap (gray)
setColorRange([10,70])
```

Plotting Histograms and Scatter plots with grain data

Sometimes we want to summarize data to show trends. A histogram will sort the data into 'bins' based on the numeric value and then count the number of points in each bin. These are great for showing how the data is distributed. On the other hand, Scatter plots are useful for showing trends (e.g. linear, exponential) between two parameters. Examples of both types of plots representing EBSD grain data are covered in this section.

Histograms of misorientation

The `plotAngleDistribution` function takes misorientations as an input and provides a shortcut for creating a histogram of this data. This simplified plot command is good for quick views of data, and most presentation options are set by default but can be added later.

```
% load some data
mtexdata forsterite
%calculate grains to get misorientations to grain mean (mis2mean) added to
the ebsd variable
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);
%copy the misorientations to grain mean for the forsterite phase
mori = ebsd('Fo').mis2mean

% plot a histogram of the misorientation angles within each grain.  Note that
this is a specialized histogram function, that accepts misorientations as an
input.
figure;plotAngleDistribution(mori)
title('Misorientations to mean for forsterite grains')

%or another example.
figure;plotAngleDistribution(grains.boundary('Fo','Di').misorientation)
title('Misorientations across Forsterite-Diopside grain boundaries')
```

Instead of using the `plotAngleDistribution` function as a shortcut, we can work manually from the grain boundary properties. Note that unlike the previous example we don't pass a misorientation variable, but instead calculate the misorientations in degrees, and pass the resulting scalar array. Since this isn't explicitly for plotting angles (unlike `plotAngleDistribution)` you'll need to name the axes manually with this method. The usual MATLAB commands can be used to adjust bin intervals etc.

```
BNDmori = grains.boundary('Fo','Fo').misorientation.angle./degree;
%since this isn't explicitly for plotting angles, you'll need to name the
axes manually with this method.
figure;histogram(BNDmori);
xlabel('Misorientation angles in degree')
ylabel('number of grains')
title('Misorientations across Forsterite-Forsterite grain boundaries')
```

Plotting histograms of scalar values

If you're working with scalar values such as grain sizes, you can just use the `histogram` or `hist` function (which result in slightly different presentation). Note that the default binning and presentation varies between methods. This syntax is good for any scalar input.

```
%plot a histogram of grain area
figure;histogram(grains.area);
xlabel('Grain area in um')
title('using the histogram command')

figure;hist(grains.area);
xlabel('Grain area in um')
title('using the hist command')

%if you input the grains parameter into the histogram functions, the default
output is a much nicer grain area percentage plot, with all axes pre-labeled.
figure;histogram(grains);
%or plotted differently
figure;hist(grains);
```

Scatter plots of various variables

For this function, the first input is the X axis and the second input is the Y axis. The third input is the circle/point size which can be another parameter of the data or an integer. Using the argument 'filled' fills in the circles. To change the marker to a dot, replace the size with '.'

Example; use the information to generate a scatter plot, with the grain area (in um) on the y axis, and the misorientation on the x axis

```
%see if grain orientation spread is correlated to grain area.
figure; scatter(grains.GOS./degree, grains.area, 2);
xlabel('Grain orientation spread in degrees')
ylabel('Grain area in scan units')
```

```
% or we can scale the markers by the area.  Since the grain size for the
forsterite dataset is large, scale the circles down by dividing by 10000.
figure; scatter(grains.GOS./degree, grains.area, grains.area/10000);
xlabel('Grain orientation spread in degrees')
ylabel('Grain area in scan units')

%or scale the markers by the result of a calculation
figure; scatter(grains.aspectRatio, grains.area,
70*grains.area./max(grains.area));
xlabel('Grain aspect ratio')
ylabel('Grain area in scan units')

%Don't forget you can use the spatial data for plots.  One thing to watch out
for, your map may have x positive to the left.  In that case, put a negative
in front of the x wherever it appears
[x, y]=centroid(grains);
figure; scatter(-x, grains.area, grains.area/10000);
xlabel('Grain centroid horizontal position');
ylabel('Size of grain in scan units');
```

We may also want to create scatterplots using misorientations. As an example of this, let's plot degrees of misorientation between a reference angle and one point per grain. In any case where you want one orientation per grain, you can use the `grains.meanOrientation` Now we need to choose a reference angle to compare these mean orientations to so we can calculate a misorientation. This reference could be a point we select (see), it could be an Euler angle we define (see) or for this example we'll use the mode of the entire phase (the most common orientation present in the whole map for the phase of interest). To do this we first use the function `calcDensity` (formerly known as calcODF), which calculates a continuous orientation distribution equation that represents the orientations weighted by frequency.
At this point, we want to get the highest intensity of this collection of all orientations. For this we use the function `calcModes`, which calculates local maxima of the odf. The two arguments input to `calcModes` are the odf we just calculated, and the number of maxima we want (which are ranked in order of decreasing maximum intensity). The outputs are the orientation of a maxima, and the intensity in units of 'times random texture'.

```
%load some data
mtexdata twins
%calculate the density function (see details above).
odf=calcDensity(ebsd('Magnesium').orientations,'halfwidth',10*degree);
%select the highest local maxima (mode) from the odf by using the value 1. We
only want an orientation from this function, so skip the return of the
intensity by using ~ in that place
[modes, ~] = calcModes(odf,1);

%Calculate the grains to get the grain.area and grains.meanOrientation
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);
```

```
%Determine the angular relationship between each grain's mean orientation and
the reference orientation
%get the orientations of the grains
ori1=grains.meanOrientation;
%calculate the misorientations on moving from ori2 to ori1
mori = inv(ori1) * modes;
%determine the angle of rotation in degrees
theta=mori.angle/degree;
%plot the data
figure; scatter(grains.area, theta, 30);
xlabel('grain area in scan units ');
ylabel('misorientation from main texture component in degrees');
```

Let's get fancy and color the circles by orientation. For this, we need an orientation colour map. See for more on how this part of the code below works.

```
%we need to define how to convert orientations to colours, so we'll use a
default function for that.
oM = ipfHSVKey(grains);
%convert the modal orientation into a 3D vector
dir = modes.*Miller(0,0,0,1,ebsd('Magnesium').CS);
%and use this vector as the inverse pole figure colour map direction
oM.inversePoleFigureDirection =dir;
color = oM.orientation2color(grains.meanOrientation);
scatter(grains.area, theta, 30,color,'filled');
xlabel('grain area in scan units');
ylabel('misorientation from main texture component in degrees');
```

Adding a trendline to a scatter plot

Let's add a trendline to the previous scatter plot.

```
%calculate the fit parameters.  The 1 here is the degree of polynomial, so in
this case, a line.
my_poly=polyfit(grains.area, theta,1);
%input a range of X values to calculate points to draw on your plot.
Grains.area runs from 0.09 to 182, so some good values for x coordinates of
your trendline might be
X= 0.1:10:180;
% use those values and the fit function to calculate matching y coordinates
Y=polyval(my_poly,X);
hold on
plot(X,Y);
hold off
```

Scatter Plot of how the misorientation to grain orientation spread changes across an EBSD map

If your map has x positive to the left. In that case, put a negative in front of the x wherever it appears as shown below.

```
[x, y]=centroid(grains);
figure; scatter(-x, grains.GOS, 30, '.');
xlabel('Grain centroid horizontal position');
ylabel('GOS in radians');
```

now add a trendline. To find the values to use for X2, use `min and max(-x)`

```
%calculate the fit parameters.  integer is degree of polynomial
%(i.e. 1 for linear)
my_poly=polyfit(-x, grains.GOS,1);
% input your X data range to calculate points to plot
X2= 1:2:50;
% use those values and the fit to calculate matching y coordinates
Y2=polyval(my_poly,X2);
% plot the trendline
hold on
plot(-X2,Y2);
hold off
```

Preparing your map for presentation

Some considerations when preparing a plot are below. There are always compromises (for example EBSD maps are only usable in colour) but do your best to balance these concerns.

- Are the text, points, and lines large enough and contrasting enough to show up clearly?
- Can this plot be understood in black and white or by someone colourblind?
- Are different data series on a line plot distinguishable from each other? Different shapes for the points will help – different colour isn't enough.

Setting map colour options

The colourbar

This is a scale that appears to the right of any map created with a linear scale. To show or hide this scale bar/legend use:

```
mtexColorbar
```

The colorbar will default to parula for any data that is scalar (i.e. this `mtexColorMap` does not apply to orientations) but the colourmap is easily changed with the code below. To see all the colour options, type `mtexColorMap` the space bar, and then tab key.

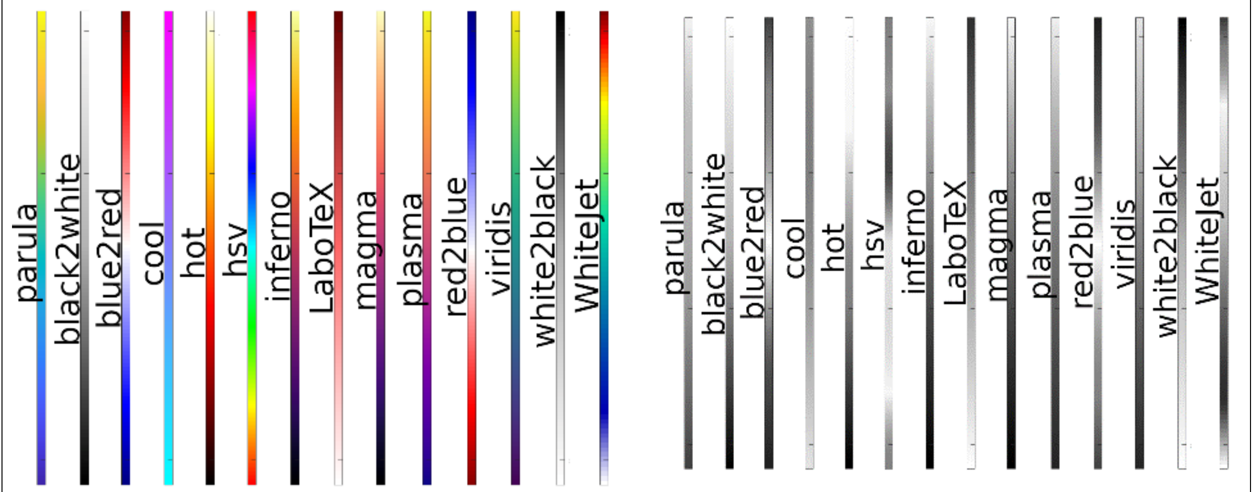


Figure 36: (left) the MTEX default colormaps (right) converted to greyscale. Parula would work in greyscale, but inferno or magma would be a bit better.

```
%to plot a map, coloured by band contrast
plot(ebsd('Magnesium'),ebsd('M').bc)
%change the color map: note this applies to active figure only
mtexColorMap magma
%you can also use matlab colormaps
colormap gray
colormap (jet)
%for inverted colormap, use
colormap(flipud(parula))

%to change the max and min values used for the colormap
setColorRange([5, 200]);
```

Additional MATLAB default maps are available and can be viewed on the following website. https://www.mathworks.com/help/matlab/ref/colormap.html You can also manually define the colourmap using RGB values, which is explained on the same website.

Setting the default phase colours

In MTEX, each phase is assigned a colour. The first phase loaded is given a light blue, the second phase a green colour, the third orange etc. You may change these colours as follows:

```
%load the forsterite dataset
mtexdata forsterite
plot(ebsd)
%view the colour of the forsterite phase
ebsd('f').color
%change the colour
ebsd('f').color=[0.1 0.1 0.3]
```

```
%plot the map again
plot(ebsd)
%change the colour to one of the MATLAB's default
ebsd('f').color=str2rgb('salmon')
plot(ebsd)
```

If you change the colour of a phase using the code above, the colours used for the grains will also update to match, so the ebsd and grains data will be coloured the same. You can also direct colour changes to the `grains` variable, (e.g. `grains('f').color=[1 0.1 0.3]`). In which case the ebsd default colour for that phase will also update.

Annotating and labeling maps with text or markers

Points can be annotated on the map on coordinate x,y using `text(x,y,'P')`.
To remove text annotations, use

```
labels=findobj(gca,'Type','Text');delete(labels);
```

You can also overlay another plot with markers on the map using the code below, where x and y are spatial positions. So for example, you can mark some points on the map using the `ginput` command.

```
%Get some points
[x,y]=ginput(2);
%To overlay the endpoints on the map
hold on; plot(x, y, '.r', 'MarkerSize',60);hold off
```

Or, if you had a line plot, you can overlay that on the map just as easily.

```
%To overlay a line on the map
hold on; plot(x, y, 'LineWidth',2);hold off
```

The colour bar can be removed/added by the command `mtexColorbar`, or by using the menu (MTEX>annotations) as can the coordinate markers.

```
%to hide the legend, first retrieve the name
hLeg = legend('example')
%then turn it off.
set(hLeg,'visible','off')
```

To change the title bar (i.e. the window header) use the code below. You can also use a variable for this, just pass it to `sprintf` the same way.

```
str=sprintf('Merged ID');
set(gcf,'name',str,'NumberTitle','off');
```

Using Crystal Shape

This function has several options, and a few built in crystal shapes (olivine, quartz, garnet etc.) Of course, HCP and cubic shape templates are included. However, with HCP unit cells, the relative dimensions can vary (some unit cells are shorter and wider than others). For example:

```
mtexdata titanium;ebsd.CS
```

gives values of 3 and 4.7 for the lengths of the a and c axes respectively, while

```
mtexdata twins;ebsd.CS
```

gives values of 3.2 and 5.2 for the same parameters in magnesium. These changes affect many aspects of the material's behaviour, in particular twinning modes, but MTEX uses the values to customize the HCP unit cell drawn with the `crystalShape` command.
Therefore we need to pass the crystal symmetry data to the crystal shape command, so that the shapes are drawn in the correct aspect ratio. Keep note: **CS represents crystal symmetry, while cS represents crystal shape**. Case is important here!

```
%load some data
mtexdata twins;
%pass the crystal symmetry from the ebsd data to the crystalShape
command, and specify that this is a HCP unit cell
cS = crystalShape.hex(ebsd.CS)
%plot the crystalShape unit cell.
plot(cS)
```

Typically, we're using this function because we want to overlay the shapes on a map. Obviously we're not going to plot one crystal shape per EBSD point, usually we want one point per grain. So let's plot a map of the grains to start.

```
%calculate the grains
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);
%Plot a grain map, using one orientation per grain.
Figure;plot(grains,grains.meanOrientation)
hold on
```

For each set of EBSD data, you're going to need to figure out the correct scale of crystal to use. Let's guess a starting value. Also, this plot can be a slow process if you have lots of grains, so let's pick one grain with the cursor, and note down the grain ID that shows in the tooltip; in my case the grain ID selected was #23.

```
%isolate your test grain
testGrain=grains(grains.id==23);
%pick a scale to start
scale =10
%overlay the grain
plot([testGrain.centroid,scale] + testGrain.meanOrientation* cS *
scale)
```

For the plot syntax above: the square braces provide the xyz coordinates. For example, `testGrain.centroid` returns the xy coordinates of the centroid of the grain, while the z parameter is provided by the `scale` variable. A non-zero Z value prevents the crystal shape from being drawn partially through the map surface (gives an 'elevation' to the shape drawing), and we can just use the `scale` variable for this since the wider the drawing is, the higher up it should be as well. The second term of the plot command (not inside square brackets) provides an orientation for the shape, which shape to use, and the `scale` variable again, this time for size.

Troubleshooting: if nothing seems to happen, increase the value used for `scale` drastically. If the map vanishes and only a unit cell shows, decrease the `scale`, if your unit cell is 'submerged' in the map surface, the z value of the plot coordinates (last part inside the square brace) needs an increase.

```
%once scale looks good, overlay the whole map.
%first, to simplify the command, break out the xy data.
GrainCoords=grains.centroid
%and add the z coordinates
GrainCoords(:,3)=scale
%plot the overlay
plot([GrainCoords] + grains.meanOrientation* cS * scale)

%note that if you decide to change the scale value, you must update
GrainCoords first.
scale=5
GrainCoords(:,3)=scale
figure;plot(grains,grains.meanOrientation)
hold on
plot([GrainCoords] + grains.meanOrientation* cS * scale)
```

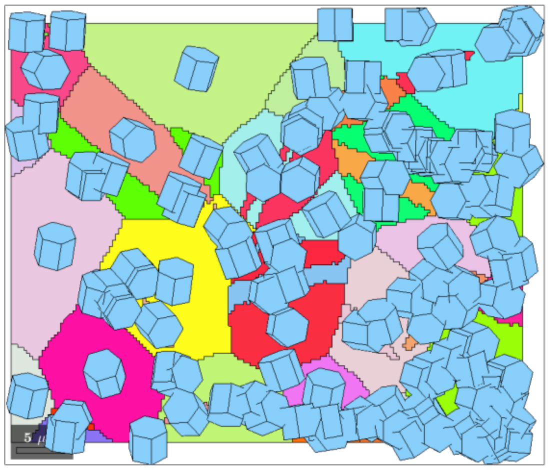


Figure 37: Crystal shape overlay on the `twins` *dataset. Scale factor of 5*

SOME LIMITATIONS: This approach will locate the crystal shape drawing over the grain centroid. If you have a grain shaped like a letter 'c' this centroid might be outside the actual grain boundaries. Also, this will generate a crystal shape for every grain, no matter how small. Also, if your grain has a wide angular range the mean orientation may not be representative of the actual orientation at the location shown. If the crystal shapes extend outside the map (which is probably the case) then a white boundary will appear to pad the map region.

Example: Crystal shape scaled with grain size

Tiny grains deserve tiny overlays (see Figure 38), so let's scale the crystal shapes with the grain size. So the xyz coordinates are set up the same, the only change is the plot command itself, where the grain area (different for each grain) is used instead of `scale` Note that you could just as easily use perimeter, aspect ratio or a similar grain parameter. Also note that you will still need to add a scale factor to adjust the sizes up or down based on the parameter chosen.

```
%continuing on from the previous example,
figure;plot(grains,grains.meanOrientation)
hold on
plot([GrainCoords] + grains.meanOrientation* cS * grains.area*0.05)
```

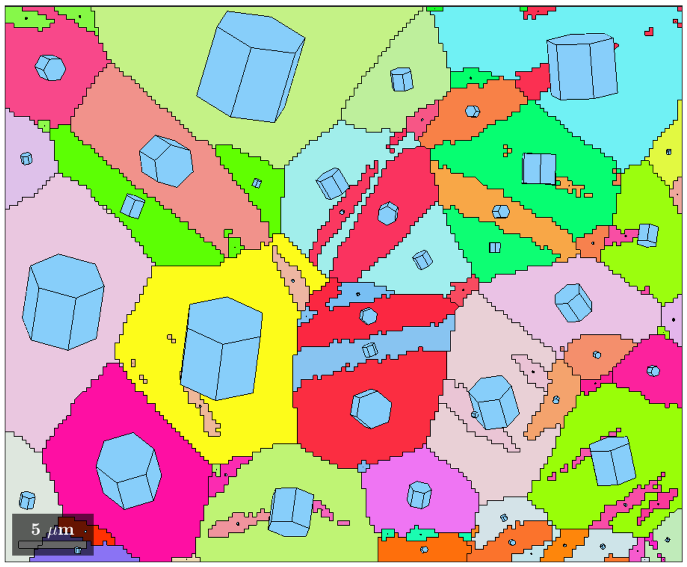


Figure 38: Crystal shapes for the `twins` dataset, scaled by area, decreased by a factor of 0.05. Code: `plot([GrainCoords] + grains.meanOrientation* cS * grains.area*0.05)`

Example: Crystal shape for only grains meeting a condition.

It's also easy to filter your data. First, let's filter based on grain size.

```
%determine a good minimum size to use
max(grains.area)
min(grains.area)
%I'll pick 100 as my threshold and filter the grains accordingly
threshold=grains.area>100
%use this new logical to filter your grains datasets
bigGrains=grains(threshold)
%for the twins dataset, this leaves me with 5 grains.  Get the xy data
and append z values
GrainCoords=bigGrains.centroid
%and add the z data.  Based on earlier plots, I know that the crystal
shapes will be scaled down by a factor of 0.05, so I need a value
larger than this.
GrainCoords(:,3)=bigGrains.area*0.1
%plot a map you can overlay
figure;plot(grains,grains.meanOrientation)
hold on
%plot the crystal shapes
plot([GrainCoords] + bigGrains.meanOrientation* cS * bigGrains.area*0.05)
```

Example, Crystal shape Forsterite data, multiple phases, filtering by orientation

Now, let's do an example filtering based on orientation. For this example we'll use the dataset named `small`. Note that you can only calculate misorientations within a single phase.

```
%load a tiny segment from the forsterite dataset, and calculate grains
mtexdata small
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'threshold',10*degree);
%get the largest grain.  In this line you simultaneously find the grain with
the largest area, create a logical of grains with that area (should only be
one) and then use that logical to isolate that specific grain.
refGrain=grains(grains.area==max(grains('f').area))
%Calculate the misorientation between the selected grain and all forsterite
grains in degrees
mori=angle(refGrain.meanOrientation, grains('f').meanOrientation)./ degree
%isolate the forsterite grains with a logical containing all grains within 20
degrees
fosGrains=grains('f')
SelGrains=fosGrains(mori<20)

%now we have our (in this case five) selected grains.  What crystal shape
should we use for forsterite?  Unfortunately, unlike hexagonal, this one
```

```
isn’t built in.  The parameters below were taken from the ‘Advanced crystal
shapes’ tutorial on the website.
cs=fosGrains.CS
N = Miller({0,1,0},{0,0,1},{0,2,1},{1,1,0},{1,0,1},{1,2,0},cs)
dist = [0.4, 1.3, 1.4, 1.05, 1.85, 1.35];
cS = crystalShape( N ./ dist)

%Now we need xy coordinates for our crystal shapes
GrainCoords=SelGrains.centroid
%add a z value so they don’t clip through the map
GrainCoords(:,3)=SelGrains.area*0.1

%note that we use the truncated grain plot command below, because
plot(grains,grains.meanOrientation) would fail as this data has multiple
phases.  The command below is a shortcut for this kind of data.
figure;plot(grains)
hold on
plot([GrainCoords] + SelGrains.meanOrientation* cS * SelGrains.area)
%Way too large.  Scale the plot down by multiplying by a small number.
figure;plot(grains)
hold on
plot([GrainCoords] + SelGrains.meanOrientation* cS * SelGrains.area*0.001)
%much better size, but hard to see due to the same colour on the forsterite
and the phase plot.

figure;plot(grains)
hold on
plot([GrainCoords] + SelGrains.meanOrientation* cS *
SelGrains.area*0.001,'colored')

%or alternately, colour them red
figure;plot(grains)
hold on
plot([GrainCoords] + SelGrains.meanOrientation* cS *
SelGrains.area*0.001,'FaceColor','r')

%or alternately, a wireframe outline
figure;plot(grains)
hold on
plot([GrainCoords] + SelGrains.meanOrientation* cS *
SelGrains.area*0.001,'FaceColor','none','linewidth',2)
```

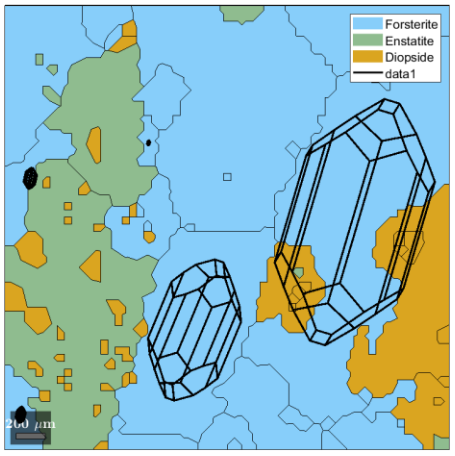


Figure 39: Crystal shape for selected grains from a forsterite phase, scaled with grain size.

Example: Crystal shape for a single twinned grain

Note that this method shows a different method of twin definition than that used in

```
% load some data
mtexdata twins
%segment the grains
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);

% copy the symmetry to a new variable
CS = grains.CS;
%Define twins using the misorientation that will map the first miller index
onto the second, and the third miller index onto the fourth.
twinning = orientation.map(Miller(0,1,-1,-2,CS),Miller(0,-1,1,-2,CS),...
  Miller(2,-1,-1,0,CS),Miller(2,-1,-1,0,CS));

% extract all Magnesium-Magnesium grain boundaries
gB = grains.boundary('Magnesium','Magnesium');

%  check which of the twin boundaries are within 5 degrees of the twinning
misorientation
isTwinning = angle(gB.misorientation,twinning) < 5*degree;
twinBoundary = gB(isTwinning)
%merge grains that share a twin boundary
[mergedGrains,parentId] = merge(grains,twinBoundary);
%plot the map to select a promising area
plot(grains, grains.meanOrientation)
%click on a promising grain that has some twins adjacent and get the number
from the tooltip.  I picked #84.  Now find which merged grain is the parent
(i.e. merged grain containing 84)
```

```
parentId(84)
%since mergedGrain #25 is the parent, we now want to isolate all the children
of mergedGrain #25.  This will include grain 84, and several other grains
that are twins.
grainPlot=grains(parentId==25)
%Let's plot this
figure;plot(grainPlot,grainPlot.meanOrientation)
```

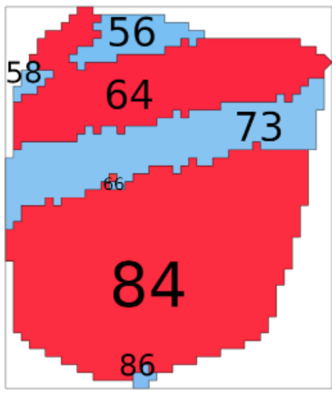
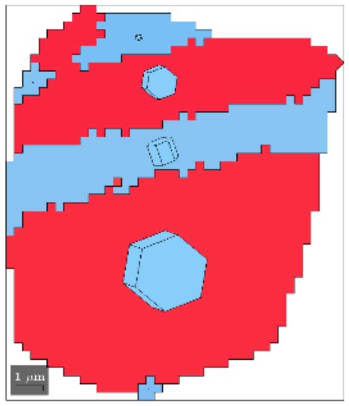
	

Figure 40: Merged grain #25 and its seven component grains. Some of these are as small as one pixel.	*The same plot as at left, but with the crystal shapes overlaid.*

```
%define the crystal shape to use
cS = crystalShape.hex(CS)
scale=10;
%define the xyz coordinates to plot at
grainPlotXYZ=grainPlot.centroid
grainPlotXYZ(:,3)=scale
%plot the crystal shapes on top of the existing figure.  I had to scale the
crystal shapes down by quite a bit.
hold on;plot([grainPlotXYZ] + grainPlot.meanOrientation* cS *
grainPlot.area*.05)
```

Example: Crystal shape based on EBSD data.

In this example, we plot every 1000th ebsd point, and don't use grains data at all. Note that you may need to do some error handling for this, including handling non-indexed data or multiple phases.

```
%take every 1000th point, not a random sample)
ebsd_tiny=ebsd(1:1000:length(ebsd))
%define the crystal shape to use
cS = crystalShape.hex(CS)
scale=2
```

```
crystalXYZ=[ebsd_tiny.x,ebsd_tiny.y]
crystalXYZ(:,3)=scale
figure;plot(ebsd, ebsd.orientations)
hold on; plot([crystalXYZ] + ebsd_tiny.orientations * cS * scale)
```

Example: Crystal shape along a line.

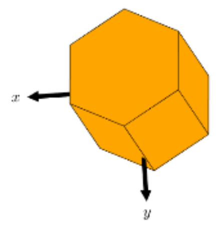
This example will plot crystal shapes along a line (e.g. a misorientation plot). Try changing the code to plot multiple selected points instead.

```
%plot a map if it doesn't exist
figure;plot(ebsd('Magnesium'),ebsd('Magnesium').orientations);
%select endpoints of a line on the map and store the x and y coordinates
[x,y]=ginput(2);
% create a variable holding the xy spatial coordinates of each end of the
desired line segment
lineSec =[x(1,:)  y(1,:); x(2,:)  y(2,:)];
%overlay the line you selected on the ebsd map as a visual reference
hold on;line(x,y,'linewidth',2); hold off;
%Send the line coordinates to the function spatialProfile, which will return
the EBSD data of points on the line as the first argument
[ebsdLineOri,~] = spatialProfile(ebsd('Magnesium'),lineSec);
%take every 20th point, so the shapes won’t overlap
ebsd_sparse=ebsdLineOri(1:20:length(ebsdLineOri))
%define the crystal shape to use
cS = crystalShape.hex(CS)
scale=2
crystalXYZ=[ebsd_sparse.x,ebsd_sparse.y]
crystalXYZ(:,3)=scale
figure;plot(ebsd, ebsd.orientations)
hold on; plot([crystalXYZ] + ebsd_sparse.orientations * cS * scale)
```

Working with pole figures-discrete points

EBSD data can be plotted as pole figures. As a simplified explanation, a pole figure is a plot showing how the crystal planes or directions plot onto a hemisphere surrounding them. This is illustrated in Figure 41. The hemisphere is then viewed from a top-down viewpoint. The crystal shape portion of Figure 41 was generated as follows, after loading the twins dataset:

```
%copy the crystal symmetry to a variable
cs=ebsd.CS
%define the crystal shape as an HCP unit cell
cS = crystalShape.hex(cs)
%define the orientation of the crystal shape
o = orientation('euler',30*degree, 30*degree, 30*degree,'ZXZ',cs);
%plot it at the origin
figure; plot(o * cS *0.9,'FaceColor','orange')
%add arrows to the figure along the x and y principal axes
hold on; arrow3d(0.5*[xvector,yvector],'labeled')
%you can then change the view angle using the menu in the figure window (MTEX> X axis
direction)
```



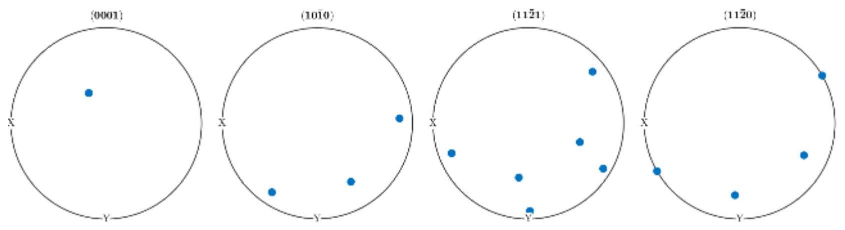


Figure 41: An orientation and corresponding pole figures. (left) crystal shape of a magnesium orientation at `o = orientation('euler',30*degree, 30*degree, 30*degree,'ZXZ',cs);`. Right: four different pole figures of that orientation.

While MTEX can also be used to import and analyze data acquired in the form of pole figures, (e.g. XRD data, neutron diffraction data) this manual only deals with data acquired and imported as EBSD information.

The minimum code for plotting a pole figure is `plotPDF(o,h);` where o is the orientation data, and h is the pole figure plane(s) to plot.

For example, to plot points from an ebsd map,

```
% load some data
mtexdata twins
%copy the orientation data into a variable
o=ebsd('Magnesium').orientations
%define the crystal symmetry, which you need to define Miller indexes.
cs= ebsd('Magnesium').CS
% load the desired pole figure planes into variable 'h' If the system is HCP,
these will show as miller Bravais indexes labelling the pole figures.
h=[Miller(0,0,1,cs), Miller(1,0,0,cs),Miller(1,1,1,cs),Miller(1,1,0,cs)];
%Plot the pole figures.
plotPDF(o,h);
```

As mentioned in the section , unless you explicitly define it any three index miller input is assumed by default to be in the form of hkl indexes, and any four index miller input is assumed to be hkil input. So the pole figure dots of `Figure 42` represent planes, and are displayed in the HKIL Miller-Bravais format. This is also indicated by the use of round parentheses used for plane normals () instead of the square brackets used for vectors [].

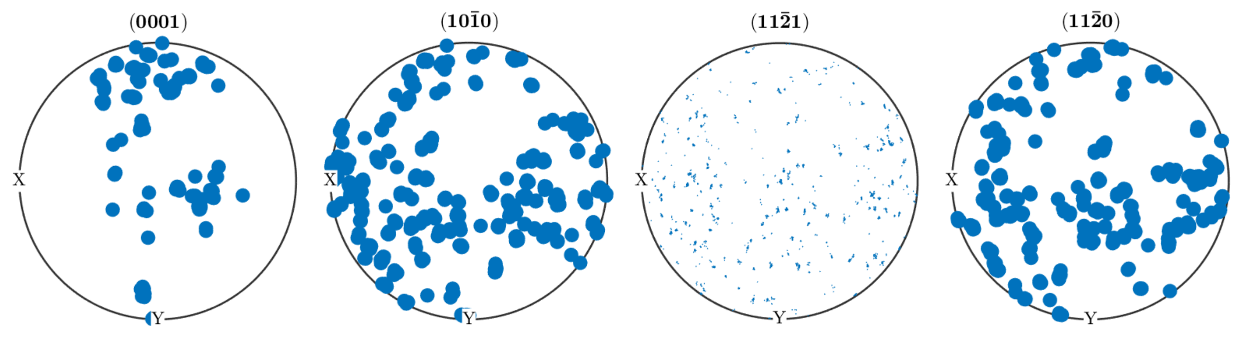


Figure 42: A series of pole figures, created with the most basic commands possible

Of course, you could also plot pole figures of the crystal vectors using the code below. In cubic systems the two definitions are equivalent (i.e. plane normals are parallel to the corresponding directions) but there are slight angular differences in HCP systems.

```
h=[Miller(0,0,0,1,cs,'UVTW'), Miller(1,0,-1,0,cs,'UVTW'),Miller(1,1,-
2,1,cs,'UVTW'),Miller(1,1,-2,0,cs,'UVTW')];
```

The pole figure command has lots of optional inputs to control what's plotted and the resulting appearance.

Unless specified, MTEX selects a certain percentage of points to plot at random from the EBSD data. It also changes the point size to be smaller when there are more points in the plot. When plotting pole figures, some **reflections** (i.e. plots of a given plane) will show more points than others. A plot of the (0001) plane gives one point per orientation, where a plot of the (10-10) plane will give three. Compare the plots of these two planes in Figure 42. Therefore the point size may be different for different pole figures unless you control it which can be done with the command `'MarkerSize'`.

Unless you explicitly state it, only a fraction of the dataset is plotted. For the example used with Figure 42, the dataset is >22k points long, but only 416 are plotted. To define the number of points, use `'points','all'` or `'points',5000` for a specific number. Remember, text gets simple quotes (not the curved quotes generated by word processing software), while numbers have no quotation mark.

A good basic code for discrete pole figures is

```
%copy your orientations for plotting
o=ebsd('Magnesium').orientations
%copy your crystal symmetry info
cs= ebsd('Magnesium').CS
%define your planes to plot
```

```
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL'),Miller(1,1,-
2,1,cs,'HKIL'),Miller(1,1,-2,0,cs,'HKIL')];
figure;plotPDF(o,h,'points','all', 'MarkerSize',1);
```

There are many more pole figure plot settings available some of which are explained below

Multiple phases, overlaying plots, and pole figures

You won't find any information here on how to plot multiple phases on the same pole figure (e.g. forsterite and diopside from the small dataset) in this manual. This is because it is essentially **not valid** to overlay different crystal structures on the same pole figure. Think about it this way: a pole figure represents how a crystallographic plane of your phase is laid out in the sample coordinates. If you have two phases with different crystal symmetry, and one point from each with the exact same Euler angle they won't have the reflections from the same plane in the same spot on the pole figure. MTEX will display an error if you try to do this.

What you **can** do is overlay two datasets with the exact same symmetry

```
%load some data
mtexdata twins
%copy the crystal symmetry to a new variable
cs=ebsd('Magnesium').CS
%define some pole figures to plot
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL')];
%Define the classic cube texture for magnesium
CubeMg = orientation.cube(cs)
%overlay these two on a pole figure (which we can do because they have the
exact same symmetry
figure;plotPDF(CubeMg,h,'all','grid');
hold on
plotPDF(ebsd('m').orientations,h,'all','grid');
```

Spherical Projection (default: equal area)

Pole figures represent a hemisphere, viewed from the apex and flattened onto a 2D view. How exactly things are flattened can be changed with options added to the plot command. To see what's happening let's look at what happens to a grid (picture a globe with latitude lines and viewed from a pole), as shown in Figure 43. The difference is clearest at the edges of the plot, showing that 5° grid lines are evenly spaced in the equal angle/stereoscopic plot created with the `eangle` argument, but compressed together in the equal area plot created with `earea`. Since `eangle` avoids compacting data at the perimeter of the pole figure it's more commonly used for materials work.

```
figure;plotPDF(o,h,'points','all','projection','eangle','MarkerSize',1);
```

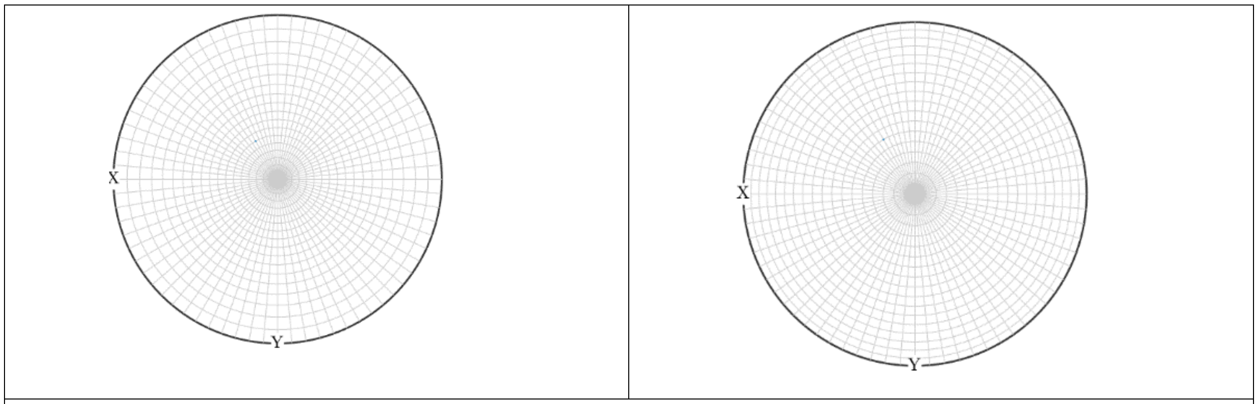


Figure 43: Two types of spherical projection (left) Equal angle (stereoscopic) projection `'eangle'` Is usually used for materials papers. (right) Equal area projection `'earea'` or `'Schmidt'`, is the default for MTEX.

Hemisphere Projected (default: upper)

You can also select which hemisphere you're looking at. Since I tend to visualize looking 'down' onto my EBSD data map, I select the upper hemisphere to view. T

```
%enter an Euler orientation
o = orientation('euler',30*degree, 30*degree, 30*degree,'ZXZ',cs);
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL'),Miller(1,1,-
2,1,cs,'HKIL'),Miller(1,1,-2,0,cs,'HKIL')];
% the plot command for all points, upper hemisphere, stereoscopic projection
plotPDF(o,h,'points','all','upper','projection','eangle','MarkerSize',1);
```

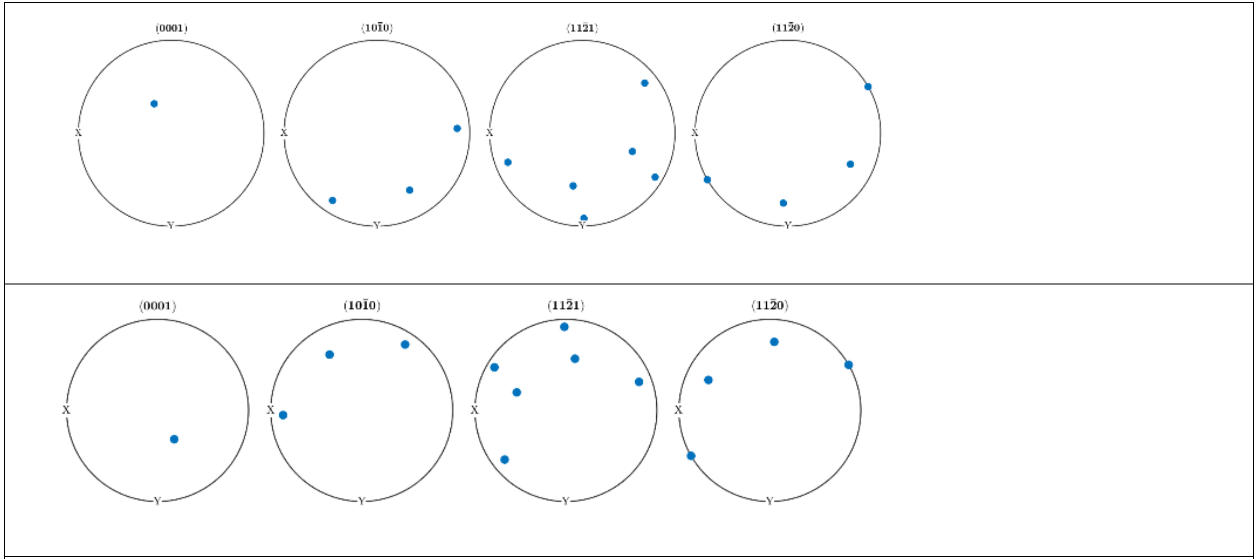


Figure 44: Effect of projected hemisphere on pole figure plot. Top: `upper` *(default) Bottom:* `lower`

Grid spacing and colour (defaults: grid off, 0.8 black)

As shown in Figure 43, you can add a grid to your plot.

```
%'grid'=adds a spherical grid
%' grid_res '= (optional argument for the grid function) sets resolution in
degrees
'grid','grid_res',15*degree,

figure;plotPDF(o,h,'points','all','upper','projection','eangle',
'grid','grid_res',15*degree,'MarkerSize',1);
```

The colour of the grid overlaid is default RGB colour [0.8 0.8 0.8] and can be edited by typing `open sphericalPlot`, editing the function, and saving the changed values. The location to edit is at lines 289 for the radial lines and 303 for the circles (or search for [.8 .8 .8]), although if file permissions won't let you save a copy you'll need to run Matlab as an administrator.

Colour of data points (default; blue)

While the default pole figure colour for discrete data is RGB 0,114,189 which is red 0%, green 45% and Blue 74%, the colour can be manually defined or a variable.

Manually defining a colour

Manually defining a colour is particularly useful if plotting multiple pole figs on top of each other using 'hold on' function.

```
%copy the ebsd magnesium orientations to a new variable.
o=ebsd('Magnesium').orientations
%copy the crystallographic symmetry
cs= ebsd('Magnesium').CS
%select some crystallographic planes to plot.
h=[Miller(0,0,1,cs), Miller(1,0,0,cs)];
%Choose a nice dark green
mb = @() [0.1, 0.5, 0.4];
%plot the figure with stereoscopic projection.
plotPDF(o,h,'upper','projection','eangle','points','all','MarkerColor',mb(),
'MarkerSize',1);
```

Colorizing PF by a property- band contrast

Points can be coloured based on a scalar property as well. See 0 for more examples. Just remember, the length of the `color` variable needs to be the exact same length as the orientation variable (which is `o` in the examples shown here).

```
%copy the ebsd magnesium orientations to a new variable.
o=ebsd('Magnesium').orientations
%copy the crystallographic symmetry
cs= ebsd('Magnesium').CS
%select some crystallographic planes to plot.
h=[Miller(0,0,1,cs), Miller(1,0,0,cs)];
color= ebsd('Magnesium').bc;
```

```
figure; plotPDF(o,h,'upper','projection','eangle','points','all',
'property',color, 'MarkerSize',1);
```

If you aren't plotting all the points, `o` and `colour` still need to be the same length. The plot command will select orientations and corresponding band contrast at random from the two variables.

```
figure; plotPDF(o,h,'upper','projection','eangle','points',200, 'property',color,
'MarkerSize',3);
```

Colorizing PF by a property- IPF

This example assumes you've got data loaded, and variables h and o already defined.

```
%generate the orientation map, converting orientation to colour
oM = ipfHSVKey(ebsd('Magnesium'))
%define the direction of the ipf coloration to use
oM.inversePoleFigureDirection = yvector;
%use the oM to convert the ebsd map orientations to a color based
color = oM.orientation2color(ebsd ('Magnesium').orientations);
figure; plotPDF(o,h,'upper','projection','eangle','points','all',
'MarkerSize',1,'property',color)
```

Size of data points

For the examples above, an integer was used for marker size. You can also easily replace the integer after the `'MarkerSize'` argument with a variable. For an example of this see .

Annotation

The default annotations for a pole figure are a label of the pole figure plane plotted at top, and the x, y sample directions (see to disable or rename the sample directions permanently).

```
%to remove all existing text on a figure, including the PF orientation
w = findall(gcf,'type','text');
set(w,'visible','off')
```

```
%to label the axes, the following inserts a square marker and a label.  The
square brackets are the vector 3D to label, the content in curly braces is
the text to put at the positions in the square brackets, and at the end is
the background colour of the text label.
annotate([xvector, yvector, zvector], 'label', {'x','y', 'z'},
'BackgroundColor', 'w');

% the following inserts a label at the selected point with no marker - note
that the text is offset due to the invisible marker
annotate(zvector,'Marker','none','label',{'R'},'BackgroundColor','w')

%for more control of the marker, the standard matlab options apply
annotate(zvector,'Marker','o','MarkerFaceColor','red','label',{'R'},'Backgrou
ndColor','w','VerticalAlignment','top')
```

```
You can also use the text command, which has no marker and positions your
label centered at the designated location.
text(xvector,'add2all','G')
```

To draw circles on the plot, you want the function below, where V is a vector3D, and the degrees defines the angular span of the circle on the pole figure. For an example of this, see . Note that because this is a vector3D (and not an orientation) it shows up on every pole figure in the same location.

```
V3=vector3d(1,1,0)
circle(V3,10*degree,'add2all')
```

Discrete Pole Figures -specific applications

Plotting a specific orientation on a pole figure

Sometimes you want to see how a known Euler angle will show up on the pole figure. In this case, the only change is that 'o' containing the orientation to be plotted is a single orientation rather than a data set. An example of this type of figure is shown in Figure 41 and Figure 44. A single orientation can be input by various means, see for some options, but typically Euler angles are used.

```
%the zxz explicitly means Bunge convention used (this is the default).
o = orientation('euler',30*degree, 30*degree, 30 *degree,'ZXZ',cs);
figure; plotPDF(o,h,'upper','projection','eangle', 'MarkerSize',5)
```

Multiple orientations on the same pole figure

Let's select two points from an ebsd map and compare the orientations on a pole figure.

```
%plot the ebsd map
plot(ebsd('Magnesium'), ebsd('Magnesium').orientations)
%get the x y coordinates of the points
[x,y]=ginput(2);
% load the first orientation into a variable
o1=ebsd(x(1),y(1)).orientations
% load the second origination into a variable
o2=ebsd(x(2),y(2)).orientations
%Finally, plot the pole figures
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL'),Miller(1,1,-
2,0,cs,'HKIL')];
figure;plotPDF(o1,h,'upper','grid','grid_res',15*degree,'projection','eangle',
'MarkerSize',10,'DisplayName','orientation 1')
hold on
plotPDF(o2,h,'upper','grid','grid_res',15*degree,'projection','eangle',
'MarkerSize',10,  'DisplayName','orientation 2')
%now add a legend
legend show
```

Pole figures of crystal directions

You can also plot crystal directions. Instead of plotting an *orientation* what you're saying is "if this point had an Euler orientation of zero, what would the pole figure of the given Miller index look like". Remember `uvtw` tag indicates a direction, while the `hkil` tag plots the plane normal. Note that if you use `uvw` tags for your Miller index, the label on the pole figure will only be three digits. See Table 4 for all options.

```
%first define the crystal symmetry;
cs= ebsd('Magnesium').CS
%then define two directions to plot
m_dir = Miller(1,1,-2,3,cs,'UVTW')
m2_dir = Miller(1,1,-2,0,cs,'UVTW')
%plot one direction, hold the figure, and add the other.
figure;plot(m_dir,'upper','labeled')
hold on
plot(m2_dir,'upper','labeled')
%Define and plot the plane normal.  Note that the plane normal is not equal
to the direction for HCP systems, and note the different brackets used to
indicate this on the pole figure.
m_plane = Miller(1,1,-2,3,cs,'hkil')
plot(m_plane,'upper','labeled')
%Plot the trace of the lattice plane corresponding to the Miller index we
just plotted
plot(m_plane,'plane','linecolor','r','linewidth',2)

%Because this plot axes align with the unit cell, we can draw that on as
well, showing how MTEX orients things by default.
cShape=crystalShape.hex(cs)
%plot the shape
plot(cShape,'figSize','small')
```

Pole figure with one point per grain

In any case where you want one point per grain, the simplest approach is to calculate the grains. Once you've done this, the `meanOrientation` variable stored in the `grains` variable becomes available, which has one entry for each grain.

```
%start with some single phase ebsd data
mtexdata twins
%calculate the grains (assuming magnesium data)
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%duplicate the crystal symmetry to a new variable
cs= ebsd('Magnesium').CS
%Select some planes for the pole figure
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL'),Miller(1,1,-
2,1,cs,'HKIL'),Miller(1,1,-2,0,cs,'HKIL')];
%plot the pole figure of mean orientations, including all points.
figure;
plotPDF(grains('Magnesium').meanOrientation,h,'upper','projection','eangle','
points','all', 'MarkerSize',4);
%title the figure
set(gcf,'name','Grain mean orientations','NumberTitle','off');
```

Filtering EBSD data before plotting

It can be helpful to filter for data quality, in this case by mean angular deviation (how much the recorded pattern varies from the ideal version in the detector library). Note that not all EBSD acquisition systems have the same data quality measures. To check what's available look at your raw data during import.

Filtering on data quality

For the `mtexdata twins` set, MAD ranges from 1.9 to zero, with higher numbers being lower quality/more questionable data (they deviate more from the mean).

```
%filter on MAD lower or equal to 1, by first creating a logical of passing
data
temp=ebsd.mad<=1;
%then load the conforming data into a new matrix
ebsdFiltered=ebsd(temp);
cs=ebsd.CS;
%Optional: check it worked, the line below should give values < or = 1
%max(ebsdFiltered.mad)
%select which pole figures to plot
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL')];
%plot about half of the data.
figure;plotPDF(ebsdFiltered('Magnesium').orientations,h,'upper','projection',
'eangle','points',10000, 'MarkerSize',1)

% optional: determine what percentage of points passed the filter
100*size(ebsdFiltered,1)/size(ebsd,1)
```

Filter for grain size

Limit data to grains with 8 pixels or more. This example assumes you've already calculated a grains variable.

```
%create a logical
temp=grains.grainSize>=8;
%filter with the logical
grainsFiltered=grains(temp);
%optional: check the result
min(grainsFiltered.grainSize)
%select which pole figures to plot
h=[Miller(0,0,0,1,cs,'HKIL'), Miller(1,0,-1,0,cs,'HKIL')];
figure;plotPDF(grainsFiltered .meanOrientation,h,'upper','projection','eangle
','points','all', 'MarkerSize',5)

%now scale the points with the grain size.
figure;plotPDF(grainsFiltered .meanOrientation,h,'upper','projection','eangle
','points','all', 'MarkerSize',grainsFiltered.grainSize)

%too big, scale them down
figure;plotPDF(grainsFiltered .meanOrientation,h,'upper','projection','eangle
','points','all', 'MarkerSize',0.5*grainsFiltered.grainSize)

%add some colour, since grain size is scalar we can use it directly
figure;plotPDF(grainsFiltered .meanOrientation,h,'upper','projection','eangle
','points','all','property',grainsFiltered.grainSize,
'MarkerSize',0.5*grainsFiltered.grainSize)
```

```
%change the colour map of the active pole figure
mtexColorMap magma
```

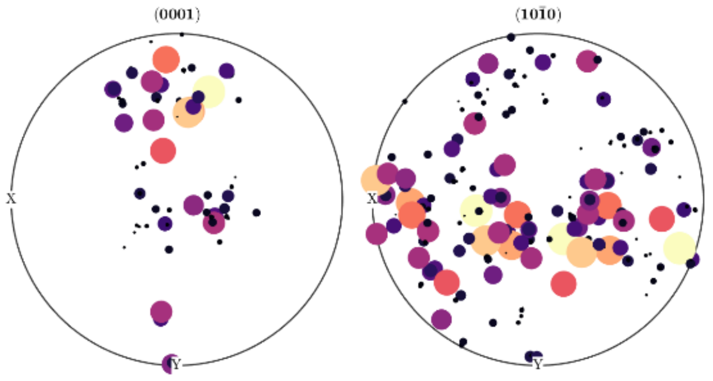


Figure 45: Pole figure of the large grains in the `twins` dataset, scaled and coloured by grain size

Pole figures with Contoured data

Contour maps are great for ignoring weaker texture components or noise, and give a sense of how intense texture components are. Compared to discrete maps, contoured data is better for comparison to XRD data and for watching how texture components change between samples. Most of the commands are the same, including upper or lower hemispheres, projection types, grids etc., see for details on those. Of course, marker colour and `MarkerSize` commands don't apply here.

To plot a contour map, it is necessary to first calculate an ODF (orientation distribution function) which is a continuous equation fitted to your discrete data (e.g. like putting a trendline on an XY plot so you can interpolate). The halfwidth of the ODF determines the smoothing of the pole figure, and thus the peak intensity. This is inversely analogous to the degree of polynomial you use to fit linear data. Too much smoothing and peaks vanish, not enough and things are noisy, see Figure 46 for an example. For each ODF, there's an associated error (the goodness of fit). Pick a halfwidth that's too high or too low, and the error increases.

So what halfwidth should we use? Well, MTEX has an automatic halfwidth selection algorithm, which takes *spatially independent EBSD* data as an input. Since EBSD data is a map, how do we make it not spatially dependent? The answer is to take one point per grain, as shown in the next example.

All values of max and min intensity are in units of 'times random'. In other words, "if you had the exact same number of measurements but the sample was a powder". Or "if the ODF was 'flat' how would this compare".

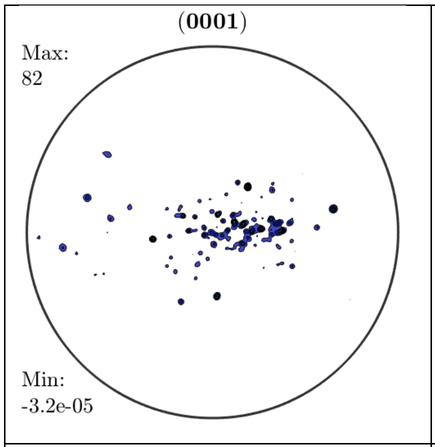
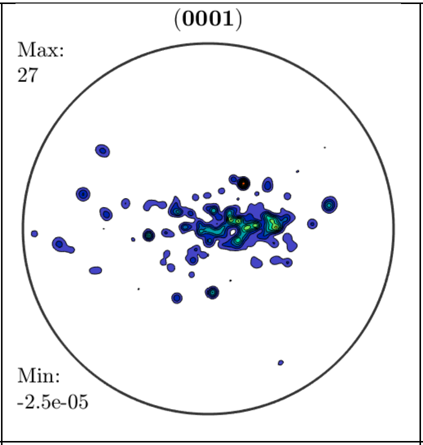
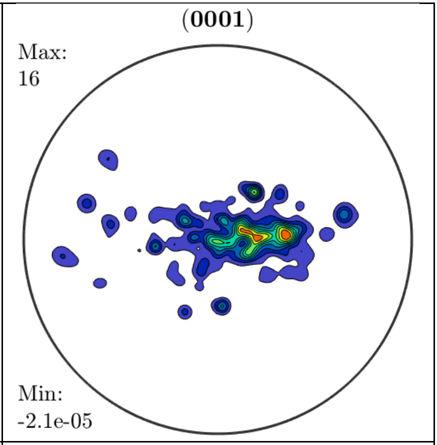
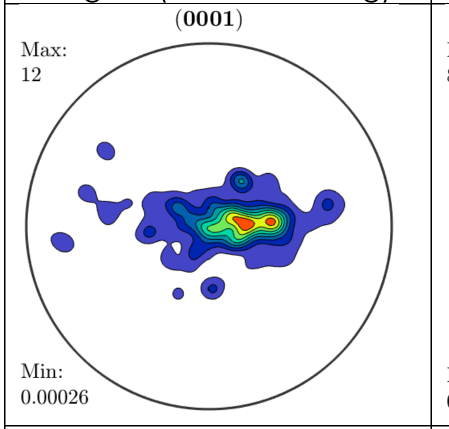
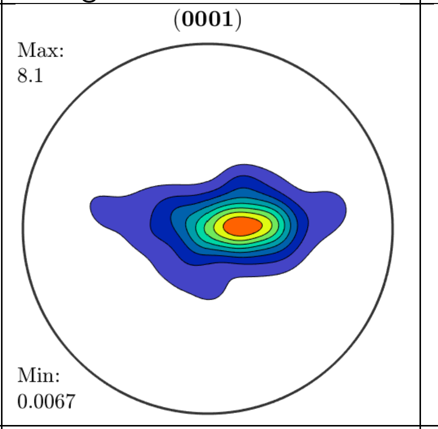
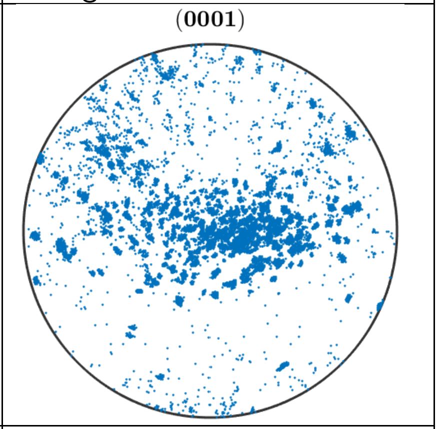
		
1 degree (little smoothing)	2 degrees	3 degrees
		
5 degrees	10 degrees (high smoothing)	The underlying discrete data

Figure 46: Six plots of the same magnesium data, a comparison of how ODF resolution affects the pole figure. Note how the maximum intensity changes.

Sample text for a continuous pole figure is below, and the output (plus some display options) are shown as Figure 47. Note that while a discrete pole figure has orientations as input, the continuous pole figures have an ODF as input.

```
%load some data
mtexdata copper
%copy the copper crystal symmetry to a separate variable
cs=ebsd('Co').CS
%calculate the grains to get one orientation per grain
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%calculate the kernel/halfwidth to use
psi = calcKernel(grains('co').meanOrientation)
%calculate the ODF
odf = calcDensity(ebsd('co').orientations,'kernel',psi)
%alternatively, define the halfwidth manually
%odf = calcDensity(ebsd('co').orientations,'halfwidth',5*degree)
```

```
%designate the cubic plane to use for the pole figure
h=[Miller(0,0,1,cs)];
%the ODF can then be plotted on pole figure.
figure; plotPDF(odf,h,'upper','projection','eangle');
```

Contour Options

There are a lot of display options, as the continuous data can be presented as a continuous colour range (the default), a set of fixed 'height' contour lines (`'contour'`), or filled contours (`'contourf'`). These are shown visually in Figure 47.

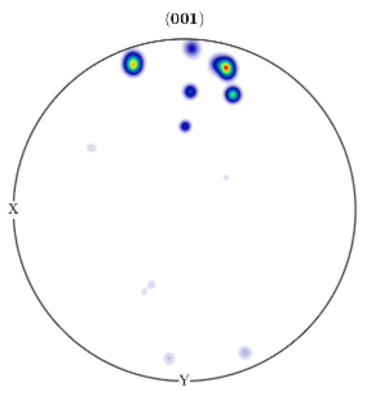
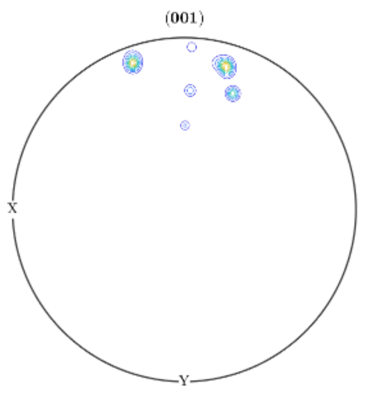
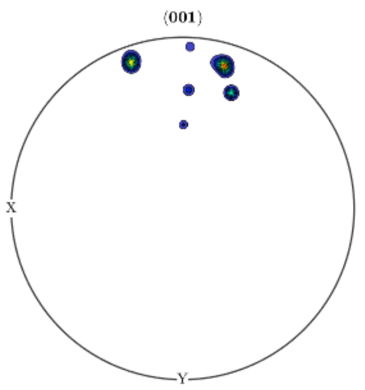
		
`%Default colouration plotPDF(odf,h,'upper', 'projection','eangle');`	`plotPDF(odf,h,'upper','projection','eangle','contour')`	`plotPDF(odf,h,'upper', 'projection','eangle', 'contourf')`

Figure 47: The `small` dataset forsterite phase ODF plotted with three display options. From left to right: continuous colour, countour lines, filled contours.

When using the contour lines or filled contours (see syntax below). There are lots of cases where you'd want control of the contour levels. For example, making sure multiple plots match up, or for comparing results to a literature result. Note that the contour settings have no effect on the min or max value displayed; that's a value directly from the ODF.

- `'minmax'` writes the min and max intensity values on the plot.
- `'contour',0.5:8 or 'contourf', 0.5:3:8` will control the max and min contour, where the first number is the min, the middle is the interval (can be omitted) and the last is the max.

```
%plot the ODF with filled contours.
plotPDF(odf,h,'upper', 'projection','eangle', 'contourf',0.5:3:8,'minmax')
```

Extracting maximum intensity from pole figures

Earlier we used the `calcModes` function to extract the maximum intensity orientation of a set of EBSD data. **This orientation will not necessarily be the maximum point of a**

given pole figure, because pole figures only show a portion of the orientation space (e.g. if you're defining a texture one pole figure isn't enough, there's too much missing information).

Below is an `example` of finding the highest intensity point on a given pole figure, using the `calcPDF` command and marking it. Note that in the example below the crystal symmetry information is drawn from the ODF rather than directly from the ebsd.CS structure, but either method is valid. Also note that you need to do one pole figure at a time, or the vector3D representing maximum intensity will be drawn in the same location on all of them which is not correct.

Keep in mind that the result returned from the `calcPDF` command is a vector3D, while the output of calcModes is an orientation. Let's compare the results by first plotting the maximum position overlaid on a series of pole figures (Figure 48).

```
%load some data
mtexdata copper
%copy the symmetry data to a new variable
cs=ebsd('c').CS;
%calculate the grains (used for halfwidth calculation)
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%calculate the optimum halfwidth
psi = calcKernel(grains('c').meanOrientation)
%calculate the ODF
odf = calcDensity(ebsd('c').orientations,'kernel',psi)
%define the first pole figure and plot it
h=[Miller(0,0,1,cs)];
figure;plotPDF(odf,h,'antipodal','projection','eangle','contourf','minmax');
%get the maximum intensity on this pole figure
pdf=calcPDF(odf,h)
[~,pos]=max(pdf)
%annotate it
annotate(pos, 'MarkerSize',10,'label','max')
%note that if the annotation does not appear, you may have to switch
upper/lower on the pole figure plot

%repeat for two more pole figures
h=Miller(1,1,1,cs);
h=Miller(1,1,0,cs);
```

Looking at the results (shown as Figure 48) note that the max is in a different position for each pole figure. In other words, **this does not provide information about the dataset, only this pole figure**. Also note that only one point is drawn per pole figure.

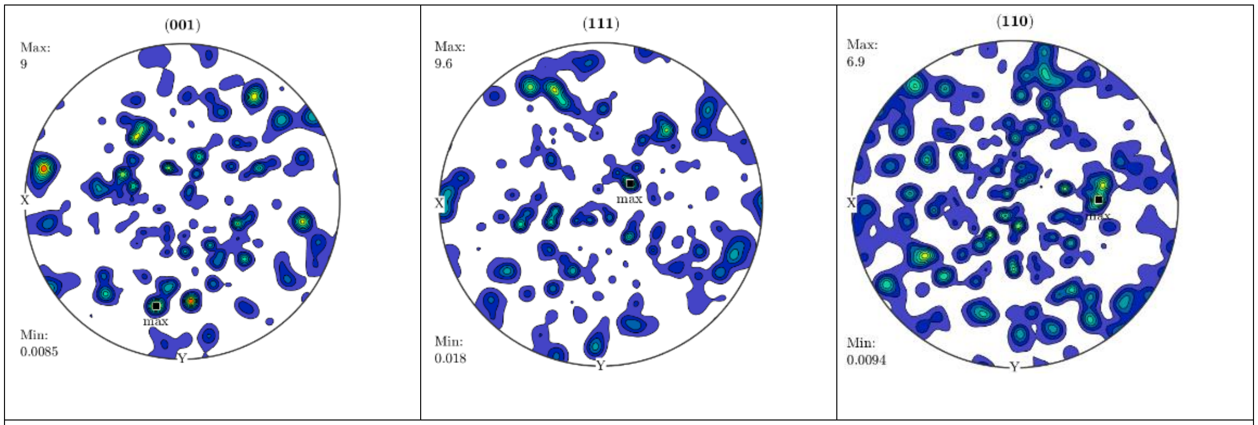


Figure 48: Copper data plotted as continuous, contour-filled pole figures, each with the maximum intensity marked

Calculate the mode and compare the results shown in Figure 49 to Figure 48.

```
%define the pole figures
h=[Miller(0,0,1,cs),Miller(1,1,1,cs),Miller(1,1,0,cs)];
%plot the pole figures
figure;plotPDF(odf,h,'antipodal','projection','eangle','contourf','minmax');
%calculate the highest intensity mode
[modes, ~] = calcModes(odf,1);
%annotate them on the figure
annotate(modes, 'MarkerSize',10,'label','mode')
```

From the output shown as Figure 49 note the mode function is near or on top of the maxima for all pole figures, but in many cases is close to the peak rather than right at the maximum value. Also note that there are three points marked in (001), four in (111), and six in (110), since we're plotting an orientation, and the number of reflections of a given orientation is different in each pole figure.

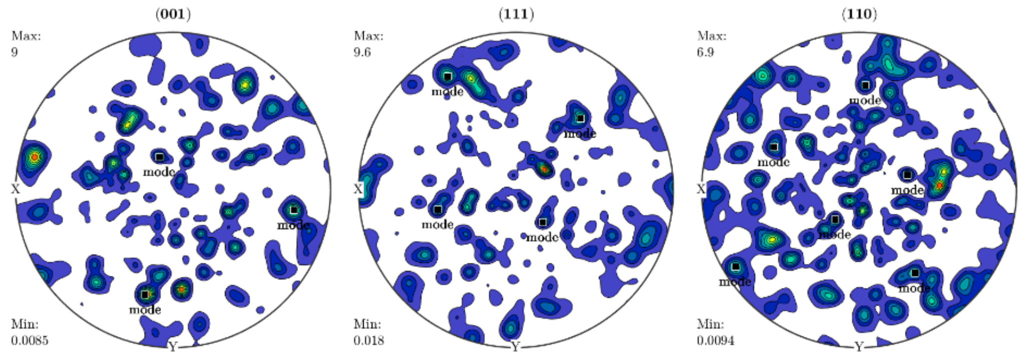


Figure 49: Copper data plotted as pole figures, each with the mode labeled

Using Modes (ODF maxima) for calculations

Since the ODF is a continuous function, it's very useful for quantifying texture in different ways. Typically this calculation uses modes (local maxima of the calculated ODF), so remember that in all cases, **the modes you find will depend on the halfwidth that was used during the ODF calculation**.

Modes (these local maxima) are located with the function `calcModes`. This function takes an ODF and the number of modes you want to find, and returns orientations from the highest intensity peak downwards. So if you enter `[modes, int] = calcModes(odf,3)`, you get three orientations listed by highest intensity downwards. If you enter two instead, you get a subset of the first result. In all cases intensity units are 'times random'. If you don't care about the intensity value, simplify the command (i.e. `[modes, ~] = calcModes(odf,3))`.

Plot histograms of angle between points and the mode

Here we will calculate the angle between all EBSD points and the highest intensity mode for the dataset, and then plot the result as a histogram, shown as `Figure 50`.

```
%load some data
mtexdata copper
%copy the copper crystal symmetry to a separate variable
cs=ebsd('Co').CS
%calculate the grains to get one orientation per grain
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%calculate the kernel/halfwidth to use
psi = calcKernel(grains('co').meanOrientation)
%calculate the ODF so we can calculate the mode
odf = calcDensity(ebsd('co').orientations,'kernel',psi)
%select the greatest intensity maxima from the odf as an orientation in the
variable 'modes', and skip return of the intensity by using ~ in that place
[modes, ~] = calcModes(odf,1);
%since we're working with angles rather than misorientations, convert to
vector3d
dir = modes.*Miller(0,0,1,cs);
%likewise convert all the ebsd orientations to vector3d
delta= ebsd('Co').orientations*Miller(0,0,1,cs);
%calculate the angles between the two, taking the minimum value in either
hemisphere
ang=angle(dir, delta,'antipodal')/degree;
%and plot a histogram
figure;histogram(ang);
xlabel('angle to mode in degrees')
ylabel('count')
```

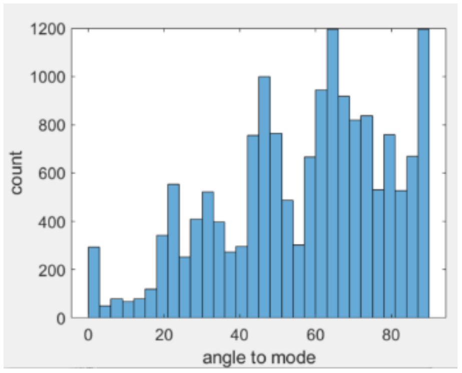


Figure 50: Angle between all data points and the mode

Determine area fractions from the map by 3D vector- c axis

This is very similar to the previous example, except rather than presenting the angle between the mode and the rest of the data as a histogram we're using a logical to count everything below a selected value and using that to calculate an area fraction. Remember, this calculates a fraction of the **original discrete EBSD points**. To make comparisons with the ODF, see .

```
%load some data
mtexdata twins
%copy the symmetry to a new variable
cs=ebsd ('Magnesium').CS
%calculate grains (required to determine the optimum halfwidth to use)
[grains,ebsd.grainId,ebsd.mis2mean] =
calcGrains(ebsd('indexed'),'angle',10*degree);
%calculate the kernel/halfwidth to use for ODF calculation
psi = calcKernel(grains('M').meanOrientation)
%calculate the ODF so we can calculate the mode
odf = calcDensity(ebsd('M').orientations,'kernel',psi)
%select the peak orientation from the odf, and input into a separate variable
[modes, ~] = calcModes(odf,1);
```

Option 1: convert the orientations to vector3D and work in angles.

```
%convert the mode orientation into a 3D vector representing the c axis
dir = modes.*Miller(0,0,0,1,cs);
%convert all EBSD orientations to 3D vectors representing the c axis
cdir=ebsd('Magnesium').orientations.*Miller(0,0,0,1,cs);
%set a threshold for your area fraction calculation
inAng=angle(cdir,dir,'antipodal')< 30*degree;
%calculate the area fraction in degrees
frac=sum(inAng)/length(ebsd('Magnesium'))*100
```

So in this case, about 50% of the orientations are within 30° of the mode for the whole dataset, assuming we use antipodal symmetry (i.e. assume any end of the 'c' axis is equivalent).

Option 2: Alternatively, you can work in misorientations rather than angles.

```
%calculate the misorientation between the mode and all other orientations in
degrees
m1= angle(modes(1), ebsd('Magnesium').orientations,'antipodal')/degree;
```

```
%calculate a logical of orientations being within 30 degrees of the mode in
degree units
inOri=m1<30;
frac1=sum(inOri)/length(ebsd('Magnesium'))*100
```

So in this case, about 28% of the EBSD data is within 30° misorientation of the mode. It makes sense that this value is lower, because option 1 effectively only compares the 'c' axis, while option 2 compares the other axes as well.

Comparing your texture against a model orientation

Once the discrete points of EBSD data have been used to compute the ODF (a continuous weighted function) this can be used to compare your data to a classic pre-defined orientation. For example, many materials have a preferred texture that forms when rolled. You can also compare your texture to one in the literature to quantify how close they are. While we did similar work above, we compared the mode to the underlying EBSD data – here we compare to the ODF instead.

You can use one of the pre-defined orientations built into MTEX, in this case, we'll compare our magnesium texture above to the classic cube orientation. You could just as easily swap `oriREF` for an Euler angle you define.

This example continues on from the one in

```
%Define the cube texture (an orientation) as a reference
oriREF = orientation.cube(cs)
%compare to the odf.  Multiplying by 100 gives us the result as a percentage
v = 100*volume(odf,oriREF,10*degree)
```

These percentages can be tracked to compare samples (e.g. successive degrees of cold rolling).

Let's look at the example above graphically.

```
%overlay these two on a pole figure (which we can do because they have the
exact same symmetry
figure;plotPDF(odf,h,'antipodal','projection','eangle','contourf','minmax');
hold on;plotPDF(oriREF,h,'all','grid','markercolor','r');
%annotate the cube textures
annotate(oriREF, 'MarkerSize',10,'label','Cube')
```

Troubleshooting

Common problems

Some things to check if you get an error message;

1. If plotting an EBSD map, check your input parameters for position and colour are the same length. In general, `ebsd('magnesium')` is not the same length as `ebsd` due to the presence of unindexed points
2. Parameter spelling and plurals are not always consistent in MTEX; `ebsd.orientations` vs `grains.meanOrientation`
3. American spelling is generally used `grains.neighbors`

4. If stuck, check out the MTEX github forum `https://github.com/mtex-toolbox/mtex/discussions`
5. For scripts, try https://gist.github.com/ search for #mtexScripts

Grid line issue

If you get blank lines periodically in your EBSD map data (horizontal or vertical) this is the result of a rounding error in the x-y coordinates of your data points which occurs when they are converted to a regular grid by MTEX. This can be fixed only by renumbering the XY coordinates in your text file to be integer multiples of the grid spacing listed in the header of the text file.

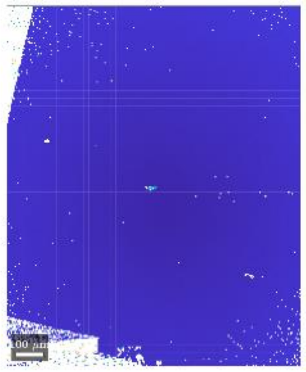
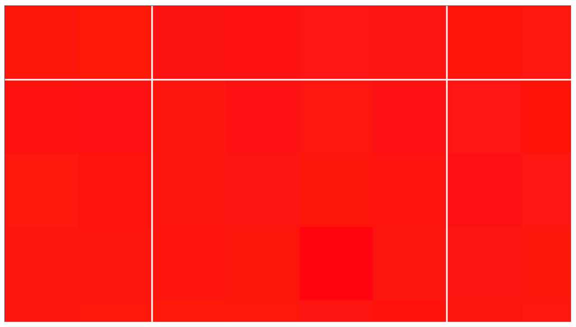
	

Figure 51: Vertical and horizontal errors in a map of a silicon chip due to rounding errors	*Figure 52: A magnified view of the spacing error.*